Títol: ACD: A common OpenGL and Direct3D driver for the Attila GPU

Volum: 1

Alumne: Vicente Escandell Miguel

Director/Ponent: Carlos González/Agustín Fernández Jiménez

Departament: Arquitectura de Computadors
DADES DEL PROJECTE

Títol del Projecte: ACD: A common OpenGL and Direct3D driver for the Attila GPU

Nom de l'estudiant: Vicente Escandell Miguel
Titulació: Enginyeria informàtica
Crèdits: 37.5
Director/Ponent: Carlos González / Agustín Fernández Jiménez
Departament: Arquitectura de Computadors

____________________________

MEMBRES DEL TRIBUNAL (nom i signatura)

President: Roger Espasa Sans
Vocal: Fernando Orejas Valdés
Secretari: Agustín Fernández Jiménez

____________________________

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:

Data: 22-feb-10
Contents

1. INTRODUCTION 6
   PROJECT CHARACTERIZATION 6
   PROJECT ORGANIZATION 8

2. BASIC CONCEPTS 9
   BASE GRAPHICS CONCEPTS 9
   BASIC GRAPHIC STACK 34
   3D GAME ENGINE 34
   GRAPHIC API 35
   DRIVER 37
   GPU (GRAPHIC PROCESSING UNIT) 37

3. ATTILA 40
   WORKLOAD GENERATION 40
   PLAYER 42
   API 43
   DRIVER 44
   SIMULATOR 45

4. MAIN CHALLENGES 48
   INITIAL PROBLEM 48
   POSSIBLE SOLUTIONS 50
   PROJECT PROPOSAL 52

5. ACD: INTERFACE 54
   INTERFACE CHARACTERISTICS 54
   HOW THE ACD INTERFACE IS DESIGNED 56
   RESOURCE MANAGEMENT 60
   RESOURCE 60
   ACDBUFFER 61
   ACDTEXTURE 62
   ACDCRENDERTARGET 65
   ACD_SHADERPROGRAM 66
   STATE MANAGEMENT 67
   ACDDEVICE 68
   ACDSTREAM 69
   ACDCREATERIZATIONSTAGE 72
   ACDSAMPLER 74
   ACDCZ_STENCILSTAGE 78
   ACDCBLENDINGSTAGE 80
### 6. ACD: SPECIFICATION

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERNAL DESIGN</td>
<td>88</td>
</tr>
<tr>
<td>RESOURCE STRUCTURE</td>
<td>91</td>
</tr>
<tr>
<td>MEMORY Object</td>
<td>91</td>
</tr>
<tr>
<td>ACDBufferIMP</td>
<td>94</td>
</tr>
<tr>
<td>ACDTextureIMP</td>
<td>95</td>
</tr>
<tr>
<td>TEXTURE ADAPTER</td>
<td>96</td>
</tr>
<tr>
<td>ACDRenderTargetIMP</td>
<td>96</td>
</tr>
<tr>
<td>ACDShaderProgramIMP</td>
<td>97</td>
</tr>
<tr>
<td>STATE STRUCTURE</td>
<td>97</td>
</tr>
<tr>
<td>SYNCHRONIZATION</td>
<td>97</td>
</tr>
<tr>
<td>OPTIMIZATION TO REDUCE THE TRAFFIC OF THE AGP BUS</td>
<td>102</td>
</tr>
</tbody>
</table>

### 7. ACD: IMPLEMENTATION

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAW CALLS</td>
<td>105</td>
</tr>
<tr>
<td>TEXTURE COMPRESSOR</td>
<td>109</td>
</tr>
<tr>
<td>CLEAR OPERATIONS</td>
<td>115</td>
</tr>
<tr>
<td>CREATE A RENDER TARGET</td>
<td>116</td>
</tr>
<tr>
<td>COPY OPERATIONS</td>
<td>117</td>
</tr>
</tbody>
</table>

### 8. AOGL: OPENGL ON TOP OF ACD

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>119</td>
</tr>
<tr>
<td>AGLEntryPoint</td>
<td>120</td>
</tr>
<tr>
<td>AGLContext</td>
<td>122</td>
</tr>
<tr>
<td>ACDX</td>
<td>124</td>
</tr>
<tr>
<td>RESOURCE STRUCTURE</td>
<td>126</td>
</tr>
<tr>
<td>AGL*OBJECT</td>
<td>127</td>
</tr>
<tr>
<td>AGL*TARGET</td>
<td>129</td>
</tr>
<tr>
<td>AGL*MANAGER</td>
<td>130</td>
</tr>
<tr>
<td>AGLTextureUnit</td>
<td>130</td>
</tr>
</tbody>
</table>

### 9. TESTING

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BATCHES AND FRAMES</td>
<td>133</td>
</tr>
<tr>
<td>RESOURCE</td>
<td>134</td>
</tr>
<tr>
<td>GPU REGISTERS</td>
<td>134</td>
</tr>
</tbody>
</table>

### 10. ECONOMIC PLANNING

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJECT SCHEDULING</td>
<td>136</td>
</tr>
</tbody>
</table>

---

* * *
1. Introduction

If you compare how games have evolved in the last few years you will get impressed of how they have improved in all the terms. Games improvements are closely related with GPU (graphic processing units) improvements. During the last years GPU flop's have rapidly being increased because of the innate independence nature of the current rasteritzation protocol used in the GPU.

Taking into account the importance that GPU's have acquire in the recent years, a group of researchers decided to study and develop high performance micro architectures for the next generations of GPU's.

To this end, a simulator and a sort of different tools have been developed. First of all, a complete cycle-accurate simulator of a GPU based on the R580, G70 and G80 generation was design and coined ATILLA.

Above this, a thin driver has been design with a goal in mind: become a thin layer that works directly with GPU transactions and memory management and allocation. Other tasks are directly delegated to higher levels of abstraction.

On top of this driver, the two most popular graphic API's, openGL and DirectX, where implemented independently from each other.

All these layers create a full 3D-graphic stack capable of running traces taken from real games, allowing us to work with real workload and obtain more realistic data from the research done on the simulator.

In order to obtain game traces from the different API's, other tools have additionally been produced, such as tools to capture openGL traces, a player to play these traces, among others.

Project characterization

By the time I started working with the ATTILA team; current API implementations are completely independent from each other. This was a reasonable decision from the point of view that in the real world each API is designed by different people without sharing among them anything. Most
commonly used are DirectX 9.0 (from now on D3D9) by Microsoft and openGL (from now on OGL) specified by the OpenGL ARB Working Group and implemented by each GPU designer.

Given the fact that many of the main features in the different graphics API are very similar, it seems quite obvious that having each different API implemented independently from each other will cause us lots of problems and more work. We will have much code that may do basically the same things but implemented many times, in fact, one time per each API we are going to support. Now, only two API’s are implemented, but in the near future, as new API’s (D3D10 and D3D11) get more popular, we will being forced to implement new API generation in order to have our workload up to date. Problems derived from this methodology will cause us an overhead because we must write much more code, debug it, and the most important one: Attila is constantly growing supporting new functionalities, forcing us to modify the entire 3D-graphic stack, and this includes modifications in each API. Too much work for the project evolved quickly.

ATTILA member take into account this problem and decided to create a graphics API abstraction layer to provide with all these common functionalities found in state-of-art graphics API’s to make easier the implementation of concrete API’s on top. This big layer actually, will be called ATTILA Common Library (from now on ACD), and it has all the basic and common functionalities across the different API’s. ACD communicates with the GPU driver and has an upper interface which provides the required functionality to allow the easy programming of specific API’s on top of it.

The ACD module provides an abstraction layer for the ATTILA Architecture, giving a friendly high-level interface of a generic 3D graphics API, suitable for building concrete API’s on top. The ACD is more than a simply HAL as usually known. It provides a lot of special features specially suited to make the graphic API implementation easier. Among them, we highlight the following features:

ACD will manage all the resources provided by the simulator, so any concrete API won’t have to manage low level data. To this end, the ACD layer will provide some data-container objects that can be used to store each object type.

GPU transactions will be also optimized. ACD will be aware of current hardware state, this includes register values and memory status, so the ACD layer will avoid useless register updates and also, data structures can be optimized before sending to the GPU, minimizing CPU to GPU bus traffic.

Another important point is that ACD will not include fixed function management. This will be delegated to a second already implemented module (as part of another student’s project) which will manage all these functionalities. This second module is called ACDX (Attila Common Driver eXtensions). The reason for separating this is because fixed function is no longer supported in
new graphics APIs due to the emergence of fully programmable GPU's which use customized programs, called shaders that replace fixed with programmable visual effects.
ACDX will be used only by older API's (OpenGL and DirectX9) in order to generate equivalent shader programs from those old fixed function states. Therefore, ACD will only work with shaders, either coming directly from the 3D application or the ACDX module.
The second goal with this project is to implement the OpenGL module making use of the ACD layer (called AOGL) to test the ACD implementation. I will use the old monolithic OpenGL module of figure 1 (full-functional and already tested) to compare with the new AOGL/ACD stack, both the visual outcome (simulator generated images) and performance (using simulator statistics).
As a result of my project, the research group will have a graphics API abstraction layer to build many others concrete graphic API's in the future.

**Project organization**

The documentation is organized as follows:

**Chapter 1:**
...

...
2. Basic concepts

Base Graphics concepts

The rendering pipeline is the responsible for creating a 2D image given a geometric description of the 3D world and a virtual camera that specifies the perspective from which the world is being viewed.

This section is going to introduce the basic graphic pipeline concepts that most of the current graphic API use. The purpose of this section is not so much to explain in depth how the pipeline works, if not to introduce the reader the basic knowledge they need to understand the whole project. In case you want to go deeper, good references are (Luna, Introduction to 3D game programming with DirectX 9.0c : a shader approach, 2006), (Luna, Introduction to 3D game programming with DirectX 10, 2008) and (Shreiner, 2010)

If we look around us we can see objects with very different shapes. Being capable of drawing all the shapes is quite difficult, because many of them don’t have regular shapes, making them difficult to be defined mathematically. Otherwise, being capable of drawing every regular shape we can imagine is also impossible because they are infinite ones. Taking into consideration all this facts and thinking how can we represent any object in the world leads us to the conclusion that all the complex objects can be built up using simple models, for example we can build a pentagon from the combination of a square and a triangle. If we go further, the most basic shape which has area is the triangle. From the triangle, practically whatever shape can be obtained.

Triangles are no more than 3 vertices connected each other, which also define a plane.

Figure 1: Face built using a triangle mech
From now on, whatever object must be split in triangle meshed. As you may probable think, using triangles to represent basic shapes works fine and is exactly the same, but when we work with curved shapes, using triangles don’t provide an accurate representation. The more triangles we use to define the curved shape the best approximation we obtain.

The triangle is the geometry shape used to define closed surfaces. Otherwise points and lines can also be used when they are needed.

Having to represent a 3D world space forces us to use a coordinate system that allows us to situate the objects in that space. Graphic theory uses the three-dimensional Euclidian space.

So, to represent a vertex there are used 3 components that define the vertex position unequivocally.

![Euclidean space](image)

Figure 2: Triangle defined using the euclidian space.

Up to now the vertex coordinates is the only attribute associated with one vertex. But many other attributes can be associated with them such as a color, material, texture coordinates, normal vector, and more.

The vertices are the most important information needed to render a game because it represents the geometry being drawn. The way how a set of vertices is transfer from the API to the GPU have changed a lot, from the prehistoric ways where a single vertex is transfer with in a single call, to the newer techniques where vertices are packed inside buffers.

New APIs use huge structures, called vertex buffers, to store the vertex information of a set of vertices without having the send them one by one. Although each API works with vertex buffers in a different way, this part tries to explain them regardless any API.
The vertex buffer is a structure where the attributes of the vertices are saved. It is possible to save each vertex attribute in one vertex buffer but also it is possible to save more than one inside an individual vertex buffer. To accommodate these buffers, the GPU hardware has added one resource called Stream where these buffers are saved and read. This unit is explained more in detail in later sections. It is necessary to take into consideration that the number of streams available is limited and depends on the hardware and also that each API creates, manages and assigns the buffers to the streams is different way.

As the information is stored inside buffers, and one buffer can contain many vertex attributes it is necessary some information to be able to read this buffers. This information is the offset to the first element, the type of the data, how many components it has and the stride between elements. Finally it is also important to know in which stream each vertex attribute is saved.

Looking closer to a triangle mesh leads us to the conclusion that many triangles share vertices with other triangles. It is possible to take advantage of this fact to reduce the information needed to specify the mesh.

The first approach is to avoid having to define the same vertex as many times as triangles use it. As in all the cases the vertex is the same it can only be defined once and used by all the triangle. With this philosophy index buffers are created. Now all the vertices that belong to a mesh are declared once. It order used to define them doesn’t matter. Now an indexed buffer it is used. This buffer contains pointers to these vertices, so instead of building the triangles from three vertices it is built using three indices that point to one vertex.

```
Index Buffer: 0 1 2 0 2 3 0 3 4 0 4 5 0 5 1
```

**Figure 3:** Example of how a triangle mesh can be represented using an indexed buffer.

Another way to decrease the number of vertices is found detecting that usually the set of triangles being drawn are next to each other. Then it is possible, instead of defining the three vertices of each triangle, to define a triangle only using one new vertex and using two of the
vertices used in the previous triangle. Then, for each new triangle it is required a unique vertex, being the only limitation that the triangle being draw is adjacent to the previous one.

There are different ways how the rest of the vertex can be choosen, but the two most commonly used are called triangle strip and triangle fan. Triangle strip obtains the other two vertices from the last two vertices specified building the preceding triangle while in triangle fan the inicial vertex used when drawing the first triangle is always used, plus the last vertex of the last triangle draw.

Triangle List
$(1,2,3,1,3,4,1,4,5,1,5,6,1,6,7)$

Triangle Strip
$(1,2,3,4,5,6,7)$

Triangle Fan
$(1,2,3,4,5,6,7)$

Figure 4: Image of how the triangles of a figure are specified using triangle list (each triangle is defined using three vertices) and triangle strip and triangle fan. For this example it is also used the indexed buffer.

In graphics, the objects are designed using a local space which is centered in the object being designed. Then each object is designed using a different space that is not compatible with the others. This is done this way because having to define an object immediately where it has to be in the world space is quite more difficult because it implies having to work with strange positions. Moreover, many times an object is used more than once in a frame, so defining the objects using the world space make it more difficult to reuse the same object, as the transformations that have to be done are less obvious.

Then, the best choice is designing all the figures using the local space, and then transforms these coordinates into the world space.
Figure 5: A square and a pyramid defined in their local space

So when we would like to use one object we must change his coordinate space, from the local space where it was designed to the world space that we are going to use to draw the whole scene. This world space will contain all the geometry we are going to draw, so they are all relative to the same coordinate system.

Figure 6: The previous square and triangle now defined using the global space.

Once all the geometry is placed in the global space is time to set the camera, from which the viewer is going to watch the scene. Camera specifies which volume of the world the viewer would be able to see and thus what volume of the world is needed to generate a 2D image of. Axonometric and perspective are the types of camera that can be defined. Perspective is the typical view that a person would have when is watching something in his real life. With this view
all the points tend to a central point where they meet. Axonometric is an artificial type of camera where the challenge is to maintain the parallelism of the lines that in the real life are parallel as well as maintain the proportion of the objects; these objectives are not achieved with the perspective camera. This camera is commonly used in CAD environments where they need to work with real proportions.

There are many ways to build up a camera, but one of the most commonly used is giving the camera position, a target point where the camera is looking to, and an up vector which points where the top of the camera is. All the points must be given using coordinates in the global space.

Figure 7: A view of an square defined using the available cameras

![Orthogonal and Perspective Cameras]

Once the camera is placed, another step is missing before the transformation is finished. We know where the camera direction is looking to, but a person doesn’t have an infinite range of view and, as in real life, the camera range of view must be bounded. The area visible by the camera is called frustum.

To bound this area there are used two different places, the near and the far planes. These two planes are perpendicular to the camera’s view direction and determine where the view’s range of view starts and where it ends. Both planes can be defined using how far from the camera they are because these planes are parallel to the xy place.

Figure 8: Falta millorar aquest dibuix, ha de ser mes complert i incloure un grafic general.
Near and far planes only bound the z coordinate. In order to do the same with the x and the y coordinates, the projection window must be defined. This window bounds the x-y coordinates and also is where the vertices are projected. The projection of the vertices is performed taking into consideration the camera type previously chosen.

In case the perspective camera was chosen, the projected points are computed drawing a line between the vertex and the origin of the camera. The intersection between this line and the projection window define where this vertex is projected. It is possible that two vertices with different coordinates are projected in the same point. Apart from saving the project x-y value it is also need to save the original z value the vertex have. This information will be used in future analysis to know how far from the viewer the point is.

The orthogonal camera works a little different, and instead of drawing a point between each vertex and the camera origin, what is draw is a parallel line to the viewer view direction that goes throw the desired point. Then the point where this line strikes the projection windows is where the vertex is projected. Here it is also necessary to save the z value.
The projection window for both cameras is defined from a view angle (alfa) and an aspect ratio. Aspect ratio by definition is $ar = \text{width} / \text{height}$. From this two values, using trigonometric operations, it is also possible to obtain how far this window is from the camera.

From all the transformations explained up to here all the vertices are expressed in the view space, which essentially is a 2D image of the scene.
Which size the projection window should have is not a trivial decision and it has its consequences when the projection window has to be map to the viewport. The viewport is nothing more than a 2D rectangle with the dimensions used by the GPU to output the final frame. So, although the aspect ratio from the projection window and the viewport can be different, it is desirable that both have the same because if not when one is map on the other, it would be necessary to readjust the aspect ratio to fit inside the viewport, deforming the original image.

Working with the view space has a problem that the coordinates are dependent to the aspect ratio used in the projection window. To make them independent from the aspect ratio another transformation is performed. This time the coordinates are transformed from the view space to the normalized device coordinates (NDC). The main difference between the view space and the NDC is that in the view space the width of the window is two times the aspect ratio and the height 2 while the NDC has a width and a height of 2.

Although we are now working with a 2D space we also need to use the Z coordinate because it is used to determine which object is in front of another. The way how each API transform the z is different. Typically the z value is scaled to a new range but this range depends on the API, and can be [0,1], or [-1,1] or [0,-1] or others. One important thing that most APIs has in common is that the scaling is not proportional, using more definition for the vertices that are near the viewer and less when they are far. This is done this way because the number of bits used to save the z value is finite, so it is possible that two vertices that are very close obtain the same z value when it is scaled. If this happen with two vertices that are very far from the viewer it doesn't matter because maybe he can see the error, but if it happens with something near the error is much bigger and can be seen.
The space transformations that have been described up to here can be set in two different ways. Old GPUs implement themselves all the space transformation process, so the API only has to provide the matrices that define the transformations from one space to the other. The main drawbacks of this system are that the transformation algorithm cannot be modified as it is implemented on hardware. This approach is also known as fixed function GPUs.

New GPUs have removed the space transformations from the hardware and they have added a new programmable unit called shader. This unit is more or less like a small processor and has its own ISA. The shader units can execute programs provided by the user. Then, now, the transformation stage is programmed by the user using the shader instructions. Then, this program is set to the GPU, who executes the code for every vertex. With this model the space transformation stage can do whatever the ISA allows to. This gives to the programmer much more flexibility as he can design its own space transformation stage. The drawback of using this system is that the API user is forced to program the space transformation stage.

To program the shader programs, instead of using the ISA, some shader languages have appeared. The most famous one are ARB and GLSL for OpenGL and HLSL for DirectX.

```
ARB
!!ARBvp1.0
TEMP vertexClip;
DP4 vertexClip.x, state.matrix.mvp.row[0], vertex.position;
DP4 vertexClip.y, state.matrix.mvp.row[1], vertex.position;
DP4 vertexClip.z, state.matrix.mvp.row[2], vertex.position;
DP4 vertexClip.w, state.matrix.mvp.row[3], vertex.position;
MOV result.position, vertexClip;
MOV result.color, vertex.color;
MOV result.texcoord[0], vertex.texcoord;
END

GLSL
uniform Transformation {
   mat4 projection_matrix;
   mat4 modelview_matrix;
};

in vec3 vertex;

void main() {
   gl_Position = projection_matrix * modelview_matrix * vec4(vertex, 1.0);
}

HLSL
```
The shader programs can access to a wide range of resources. The most important ones are the input and output parameters. Input parameters provide the data read from the streams and the output ones is where the shader program has to leave the results for the next stages. Also the shader can access to a constant table that is provided by the programmer with the shader code. This constant table is used to store some values that are used inside the shader program. How each shader access to these resources depend on the shader language used.

Another important task performed by the vertex program, or by the fixed pipeline when no shader program exists, is lighting the scene. Next part will introduce color theory needed to understand how color is selected.

Apart from performing the space transformation, the vertex shader also computes the light for each vertex. Next section explains some color theory necessary to understand this stage.

**Color theory & lighting**

There are many ways to represent a color but graphic's usually use the RGB model. RGB is the acronym for red, green and blue color, from which practically whatever color can be generated.

This model was selected because according to the trichromatic theory, the retina contains three kinds of colored light receptors, each sensitive to red, green or blue light with some overlapping. The incoming RGB light stimulates its corresponding light receptor creating the picture inside the eye.

The colors described using the RGB schema are expressed using this nomenclature (R, G, B) where each one means red, green and blue and are represented using values between 0 and 1 or 0 and 255.

Usually each component is saved using 1 byte, having 256 possible values per component and resulting in 16,777,216 possible colors.
Apart from these component, in graphic it is also defined the alpha component, being the model RGBA. The alpha component is used to define how transparent is the color, in other words, how easy is to watch throw it.

When a person looks at an object what it really sees is the light reflected by the object he is looking at. The more light the object reflect the more bright the object is seen. To test this you only have to be inside a room with an incremental switch and start incrementing and decrementing the light. As you increase the power, objects get brighter while as light is decremented they get darker until light is off when nothing could be seen because no light is reflected.

As the objects reflect the light they receive, they also act as a light source. Then, objects that are supposed to be in dark because they are not facing the light source they receive some amount of light obtained from the light that other objects reflect.

The model explained above follows the global lighting model, which faithfully represent real world lighting. But computing all the possible light interactions, not only between each light source and each object but also the light each object reflect, is so computational expensive that it cannot be achieved in real time applications.

In order to obtain credible illumination models without being so computational expensive local lighting model must be used. Local lighting model has some differences that make it much faster than the global model. While global model compute all light interactions between each light source and object and between different objects, local model only compute the contribution that each light source generate over each object. This model computes individually which is the contribution of each light source to a given object, without taking into consideration that there is more geometry in the scene. This model doesn’t take into consideration the possibility that other objects can lay between the light source and the object being considered, so in this case the object that is rear the other one and that is not supposed to receive light from the source would be illuminated as if no object is in front of it, so this model doesn’t generate shadows and if they are desired they must be generated using other techniques.

The absorption and reflection of a light ray depends on many factors, so in order to compute it credible, Lambert Cosine Law is used. It is necessary to take into consideration that when light strikes point head-on some surface the reflection is more intense that when light just glance a surface point. In order to determine the angle with which light strikes the surface, the vector product between the vertex normal, vector that describes the direction a polygon is facing to, and light direction vector would be used. So, Lambert Cosine Law expresses the intensity with which light strikes and object following this formula:

$$\text{Intensity} = \max (\cos \text{angle}, 0) = \max (L \cdot n, 0)$$
Where \( L \) is the light direction and \( n \) the normal vector. Intensity range is from 0 to 1.

As in real life can be seen, light sources also have some color. Their color is also represented using the RGB agreement, being for example the red light \((1, 0, 0)\), yellow \((1, 1, 0)\) and so on.

In the real work everybody thinks that the objects have some color. This is not really true, in fact what a person see as the color of an object is the light that it reflects and arrive the viewer. Then, the color of the object is not defined by the real color a person sees, if not by the material it is made with. The material is nothing more than a property that specifies which colors the object reflects and which absorbs. It is important to highlight that the color reflected by an object not only depends on the material it is made with but also the colors the light source has. Then, it is possible that a material reflects red and blue light, but as the light source is red, the object is seen by the viewer red because no blue light is reflected as the light doesn’t emit it. Materials are also defined using the RGB convention; in this case, the bigger each component is the most light it reflects.

Another important topic to analyze is the different light sources that can be found.

**Ambient light**

As was told before, light reflections generated by other objects are not considered in the local model. If this rule is followed strictly, every area part of an object that is not facing the light source would be entirely black. This behavior is not very realist, because as was told before, the objects that receive some amount of light work like new light sources, then is very probably that the objects that don’t receive the illumination of one of the main light sources receive the light reflected by other object. To simulate this behavior without having to follow the global model, the ambient light is set. The ambient light is nothing more than a light contribution that is added to every vertex of the scene regardless where the object is. This light is used to simulate the light reflected by the objects that receive the light from the main sources.

To use the ambient light the following formula is used in every vertex:

\[
\text{Reflected ambient light} = \text{Light}_{\text{ambient}} \times \text{material}
\]

as an example

\[(0.5, 0.25, 0) = (1, 0.5, 0) \times (0.5, 0.5, 1)\]

Being ‘\(\text{Light}_{\text{ambient}}\)’ the color of the light source and ‘material’ the material used in this vertex.

**Diffuse light**

The diffuse light appears when a ray of light strikes a surface and this ray scatters in various random directions. The fact that the light is scattered in random directions makes that no matter
where is the viewer that he would receive the same amount of light. As previous seen, when light strikes point head-on some surface the reflection is more intense that when light just glance a surface. This must be taken into consideration when it is computed how the diffuse light interacts with the object material.

Then to compute the diffuse light the most important information is compute with which angle the light source strikes the surface. This is compute using the Lambert Cosine Law.

![Diffuse](image)

**Figure 13:**

So being ‘n’ the surface normal vector and ‘L’ the light ray vector, the reflected diffuse light is computed using the following formula:

\[
\text{Reflected diffuse light} = L \cdot n \times \text{Light}_{\text{diffuse}} \times \text{material}
\]

**Specular light**

Smooth surfaces have a different behavior when a ray impacts on it. In this case the light is reflected sharply in one direction. As you can see the most important difference between diffuse and specular lights is that in one case light is scatter in all directions while in the other case the light is only reflected in a unique direction. Then the viewer can only see the light in case the vector defined by the reflected light points to the viewer position.

As it can be seen, it is very difficult that this situation take place because the viewer must be just on the reflected light to be able to see it. Another problem is that even the viewer is able to see the specular light, if he moves a little from this point the specular light will suddenly disappear. In order to make it more realist, instead of defining a reflectance vector to define which viewers can see this like, it would be used a one. With this cone more people would be able to see this light. Also, as is not the same being on just on the reflectance vector than on the cone boundary, the closer the viewer is from the cone center the more intense the light it will receive.
The size of the cone is determined by the angle between the specular reflectance vector and the side of the cone, represented using $\alpha$.

To model this behavior, the Lambert's Cosine Law must be modified adding some power $p$ to the previous formula. $p$ is the variable that connects the angle of the cone with Lambert's Law. Then, the bigger $p$ is the smaller the angle would be and vice versa, the smaller $p$ is the bigger the cone angle is.

$$K_s = \begin{cases} \max(\cos \alpha, 0) & \text{if } \mathbf{v} \cdot \mathbf{r} > 0 \text{ and } \mathbf{L} \cdot \mathbf{n} > 0 \\ 0 & \text{otherwise} \end{cases}$$

Reflected specular light = $K_s \times \text{Light}_{\text{specular}} \times \text{material}$

Usually all three types of light (ambient, diffuse and specular) are present in a scene, so the final formula modeling the three lighting models is:

$$\text{Reflected light} = (K_d \times \text{Light}_{\text{diffuse}} + K_s \times \text{Light}_{\text{specular}} + \text{Light}_{\text{ambient}}) \times \text{material}$$

Up to know the different lighting models have been explained. We have been talking how important is the angle with which the ray strikes a surface is some models, but nothing has been said about how the light sources are. Next section explains the different light sources.

**Parallel light**
Parallel lights represent sources of light that are so far away from the object that the rays arrive to the object in parallel, so all light rays are parallel to each other. An example of this kind of light is the sun light. To define the parallel light it is only necessary a vector that describes one ray as the others are parallel to this.

**Point lights**
Point lights are typical light sources that radiates spherically from a given point. Good examples of this kind of light are bulbs. In order to make them more real, light intensity weakens as a function of distance based on the inverse squared law:

\[ \text{Intensity(distance)} = \frac{\text{Light intensity}}{d^2} \]

The results obtained from the above formula are not quite good when they are used in computer graphics, so, to improve the results it is used a different one that is more configurable and with which designers can obtain better results changing the values:

\[ \text{Intensity(distance)} = \frac{\text{Light intensity}}{(a_0 + a_1 \cdot d + a_2 \cdot d^2)} \]

If this formula is combined with the formula obtained in the previous section that defines the reflected light, the following formula is obtained:

\[ \text{Reflected light} = (K_d \times \text{Light diffuse} + K_s \times \text{Light specular} + \text{Light ambient}) \times \text{material} \]

\[ a_0 + a_1 \cdot d + a_2 \cdot d^2 \]

**Spotlight**
Spotlight is more or less the same as point light but with the difference that instead of radiating to all the directions, it only radiates using a cone shape.

The manner in which spotlight is calculated is similar to specular reflection. The cone light is defined by a vector \( d \) which defines the cone center and the angle of the cone represented using maxAngle. As when the specular light is compute, the more closely the ray is to the boundary of the cone the less intensity it have. So for a given ray, that has an angle \( \alpha \) with the cone center the light intensity is defined by the following formula:

\[ K_{\text{spot}}(\alpha) = (\max (\cos \alpha, 0))^{\text{maxAngle}} \]
Spotlights also take into consideration that the farther an object is from the light source the less intensity it receives, so spotlights also uses the same attenuation parameter that have been seen with point lights. So the final formula, which also takes into consideration the reflectance formula obtained in the previous section, to compute the spotlight is the following:

\[
\text{Reflected light} = K_{\text{spot}} (K_d \times \text{Light}_\text{diffuse} + K_s \times \text{Light}_\text{specular} + \text{Light}_\text{ambient}) \times \text{material}
\]
\[
\quad a_0 + a_1 \cdot \text{distance} + a_2 \cdot \text{distance}^2
\]

As a recap, we have seen that there are different light sources and all the available way how this light source interacts with the surfaces they strike. A complex scene, with multiple light sources, for each vertex it is necessary to compute the light they receive from each light source. Then, the more complex the scene is, the more computational cost it would have.

After each vertex has crossed throw the vertex shading stage is time to group every set of vertices into the appropriate previously primitive set. The available primitives are points, lines, triangles and squares. As previously seen, the triangles and squares can be specified in three different ways: list, strip and fan.

From now on, the working set the graphic pipeline works with is the primitive.

All the transformations performed in previous stages of the pipeline don’t reduce the amount of vertices in the system. Next stage is clipping, which removes the geometry that is outside the frustum.

What clipping does is discard all the primitives that are outside the frustum area. Primitives that are completely outside the frustum are easily to discard, but there are some ones that are half in
and half out. In those cases what is done is clip the part of the primitive that is outside the frustum and create smaller primitives from the part that is inside the frustum.

Another technique that is used to reduce the amount of primitives is culling. In a scene there is some primitives that are not looking to the viewer, as this geometry can be seen by the viewer it can be discarded, and this is what the culling test do.

Figure 15:

In order to determine if a primitive is facing the viewer some conventions must be taken. First of all, the order in which the vertices are introduced in the graphic pipeline is important. To determine where the primitive is facing, the primitive normal is being computed. The normal is computed with the vector product between the vector defined by the second and the first vertex of the primitive and the vector defined by the third and the second vertex. Which is the first, the second and the third vector depends on how they are introduced. The order used to define the primitive defined

Up to now, the working unit is the primitives and they are represented using the NDC coordinate system. Next step is mapping this space to the viewport. This stage is very important because is the first one where the coordinates are discretized.

When the primitives are represented using the viewport range is time to rasterize the surface defined by the primitive. This is done inside the rasterization stage. The rasterization stage traverses the primitives, using one of the available rasterization algorithms, and generate from them some objects called fragments. These fragments are the new working unit in the pipeline and can be map to the final pixels but it is necessary to understand that fragments are pixels that may or may not be viewable in the final frame. The fragments object contains the same
attributes as the vertices of the primitive from where it has been generated, but this attributes have been interpolated between all the vertices of the primitive.

Rasterization

New stage in the fragment life-cycle is fragment shading. Fragment shading is for a fragment what vertex shading is for a vertex. As in vertex shading, old GPU implement this stage using a fixed-function structure.

The main function of the fragment shader is to generate the final color of the fragment.

The fragment shader program can access to the constant table, one different from the constant table used in the vertex shader, to the inputs of the fragment shader, which are the attributes previously interpolated in the rasterization stage, and also to the outputs generated by the fragment shader.

Apart from this, the fragment shader can also access the texture unit to sample a color from a texture. How can be accessed is explained in the next section.

Texture

Having to define a color for each vertex doesn’t provide good result if it is desired to paint complex pictures on the geometry with the basic tools explained until now, basically assigning a color to each vertex. For example, the only way to paint a photo on a primitive that is a square is creating a big mesh with hundreds of triangles and assign to each one the appropriate color to simulate the image of the photo. The quality of the photo on the square depends on how many triangles have been created, the more triangles used, the more definition it has. Of course, the main drawback of this technique is the amount of triangles needed to paint a surface, instead of using an square, it is need to use a big mesh of triangles.
One solution to this problem is using textures. A texture is nothing more than an array that contains color information, usually defined using the RGBA color space. Then, instead of assigning to each vertex a color, it is assigned to them a texture coordinate, which references a position inside the texture. This texture coordinate is very important because it references a point inside the texture, basically a color, and as texture coordinates are also interpolated after the rasterization stage, each fragment has a different texture coordinate, so a different color. Then, the texture is map on the geometry.

Texture coordinates are described using u and v vector and a range between 0 and 1 as the following picture shows:

![Texture Coordinates](image)

**Figure 16: Correct the texture es 256,256 x 1,1**

Each [u,v] point in a texture is called texel.

The texture of a door is a 2D texture but there are more texture formats. Obvious ones are 1D and 3D formats; they have 1 or 3 dimensions, so texture coordinates have 1 or 3 values. Another format is the cube map. Cube map is built using six 2D textures and they are stick together as a cube. This kind of textures is mainly used to draw environment textures as all the faces of the cube map are associated.
Until here it seems easy; there are some texture coordinates that are mapped to a texture, but there is one problem because when some texture coordinates are assigned to a triangle we don’t know how big the triangle will be on the frame buffer. Imagine a square where a texture is mapped. This texture has 16 texels but when the square is rasterized it has generated 256 fragments. Then we have a lot more fragments than information has the texture. The opposite case can also take place, in fact is strange that the number of samples of the texture and the fragments generated from the primitive are the same. Both situations represent a problem; in the first case, the texture doesn’t provide enough information (each texels is bind with more than one fragment) but in the second case there are too much information because there are some texels that are not bind with any fragment, and this can cause that the texture doesn’t seem realist.

To solve these two problem the following techniques have been proposed:

**Magnification**
In this case, the number of fragments generated from a primitive after the rasterization is bigger than the texture. The best way to solve this problem is using bigger textures. The problem of using bigger textures is that every day the resolutions of the screens increase, so bigger resolutions are required.

There are two possible solutions to hide this problem: nearest or linear interpolation.

**Nearest interpolation** samples the fragment with the nearest texel to the texture coordinates. This is a low cost method but provides the worst results.

**Bilinear interpolation** is a complex way in which the final color is obtained from combining the color of the four nearest samples to the texture coordinates. The color of these four texels is combined using a weighted average according to distance. The results obtained using the
bilinear interpolation are better than with nearest, but it has to perform 4 accesses to the texture.

![Figure 18](image)

**Minification**

Minification takes place when the number of fragments generated from the rasterization of a primitive is smaller than the texture itself. The problem is that there is too many information for the fragments, so from all the available texels of the texture some of them must be select. Imagine the same photo described before. This photo is map on a square but this square is far away from the viewer, so only a few fragments appear on the screen. Imagine that this is a photo of your face. If the texels are not properly selected it is possible that instead of seeing something familiar to your face (remember that is far away from the viewer) it is possible that the result is a black box because the texels selected are from the boundaries. The best solution here would have been to select the best texels to define something similar to a face.

The best way to solve this problem is working with smaller texture obtained from the original. These textures are called mipmap levels. Then, when the previous situation happens, instead of using the biggest mipmap of the texture, it is used one that has approximately the same amount of texels as fragments. This mipmap would be better than the previous because the colors of the texels would have been selected carefully to obtain a credible image. The size of a mipmap level is half the size of the upper mipmap level being the smallest one a 1x1 mipmap.
Mipmaps can be generated by the API itself or designed by a graphic artist. Obviously being designed by the artist provides the best result.

Now the problem is how the appropriate mipmap level is selected. To solve this problem 4 different techniques are explained:

**Nearest filter**
The nearest filter chooses the closest mipmap level to the number of fragments we are working with. Then, the appropriate fragment color is chosen using the nearest magnification interpolation.

**Bilinear filter**
The bilinear filter chooses the mipmap level using the same method as the nearest filter, but now the final texel color is selected using linear magnification interpolation.

**Trilinear filter**
Nearest and bilinear filters has an abrupt change when mipmap level changes. To solve this problem the trilinear filter selects the two closest mipmap levels. Then, to each mipmap level, the bilinear filter is applied and the result obtained from each mipmap is interpolated obtaining the final color.

**Anisotropic filter**
All the filters explained until now have one problem, if a primitive is at an oblique angle from the viewer point of view, it looks blurriness. The problem appears because the previous filters sample an square from the texture. The way to solve this problem is sample a trapezoid. Anisotropic filtering samples trapezoid of the mipmaps and perform the trilinear filtering.
When a texture has been defined, it has been said that the textures coordinates range from 0 to 1. In fact this is not true, and it is possible to use whatever positive value to define the texture coordinates. When the texture coordinates are bigger than 1 the wrapping mode is used. The wrapping mode specifies what happens when the texture coordinates are bigger than one. There are different modes but the most important ones are ‘repeat’ and ‘mirroring’. In the ‘repeat’ mode the integer part of the

Finally it is important to talk about how these textures are set in the API. How textures are set is very similar to how the vertex buffer are set. The hardware provides some units called texture units, where the textures are set. Each texture unit contains the data of the texture, all its mipmaps, as well as all the information necessary to read it, such as the minification and magnification filters, the wrapping mode, etc.

From the definition of what a fragment is, it is obvious that not all the fragments are going to be draw in the frame buffer, so what next step would try to do is discard some of this fragments to reduce the amount of fragments to process. Fragments that are not going to be displayed are those that are rear others, so techniques we are going to use try to determine if there is something in front of them. Here is where is so important how the Z values is computed in previous stages.

To perform the test we are going to explain, we must introduce a new buffer which will be used among this test, the Z-Buffer. This buffer hold for each of the pixels of the frame buffer area which is the Z value of the last value wrote into the frame buffer. So checking this buffer allows as to know how near or far is the current fragment being drawn respect to the current fragment. Z-test will allow us to discard the fragments that are not going to be drawn in the frame buffer because their Z is further than the Z that the current fragment in the frame buffer has. Z-test will allow us to discard many fragments and will reduce the among of load of our system.

Stencil test is another test that can be performed to discard fragments. This test blocks certain areas of the frame buffer to not be drawn. To be able to perform stencil test another buffer is included, the stencil buffer. This time this buffer will contain some reference value that will be used to evaluate if the fragment must be killed from the pipeline.
Stencil works applying over every fragment one formula that decides if a fragment can continue:

\[
\text{If (reference value} \land \text{mask OPT stored value} \land \text{mask)}
\]

\[
\begin{align*}
\text{Accept} \\
\text{Else} \\
\text{Discard}
\end{align*}
\]

An operation OPT is performed between the stencil reference value, value that was set throw the API, and the value that is store in the stencil mask. Over those values an ADD mask is applied. Typical stencil operation are comparison operators (\(<, \leq, \ldots\)) and they can return two possible values, true or false. In case the answer is true, the stencil buffer is updated with the reference value otherwise the fragment is killed from the pipeline. Stencil test is usually use to create shadows in a scene using shadow maps.

Scissor test is another test that can be performed to discard more geometry. The objective it aims is more or less the same as stencil buffer only paint some portions of the frame buffer. This time the way we select which area of the frame buffer is selected is more simple and is done defining two points from the frame buffer that define an square area. This area will be the only one that can be updated, other fragments that are mapped on pixels outside this area are discarded.

Another test is alpha test. Until now we haven’t say anything about the alpha color that sometimes color have. This test discard geometry based on this value. As in stencil test, user defines a reference value that is used to be compared against the alpha value that each fragment have. As in stencil test many operations can be set between them. In case the result is true, the fragment continues down, otherwise it is discarded.

After all this tests have been performed we know for sure that our fragment will be drawn in the frame buffer, but what we don’t know is how the color of the input fragment will be combined with the current color of the fragment in the frame buffer. This technique allows us to create effects such as transparencies.

Inside blending stage, fragments that are currently rasterizing are called source fragments while fragments stored in the frame buffer are called destination fragments. So the formula used into the blending stage is:

\[
\text{CF} = C_{\text{src}} \times F_{\text{src}} \text{ blending op } C_{\text{dst}} \times F_{\text{dst}}
\]

Where Csrc and Cdst are the colors from the fragments and Fsrc and Fdst are the factors of each color we are going to use. Blending operation available depend on the graphic API itself but commonly operations are src color, dst color, add, subtract, min, max, among others.
When the fragment exits the blending stage it has his final color and can be written to the frame buffer, updating also the Z-Buffer with its Z value.

// Imatge sobre el pipeline

**Basic graphic stack**

Between the first simple games programmed to run over a CPU which only have thousands lines of code, to current complex games which uses much more resources than many other task, with huge detailed levels and characters which appear real, basic graphic stack has changed a lot in order to be able to support high demand graphics.

Over the years, game programming have changed so much, not only for the increasing difficulty and complexity they have, but also for the need to reduce how much it costs to program a game. As always in computers to make it easier to do something, levels of abstraction have been added to simplify the work of creating new content. That’s the reason why game graphic have evolved in such a way.

As we previously talk about, the basic unit for games is the GPU, which manages all the low level operations to create games. Over this piece of hardware we find a driver which controls the hardware itself, being this the first level of abstraction. Over this driver we find one or more API’s attach to it. Basically graphic APIs are the first abstraction which provide real drawing primitives, and is the first way game programmers can start creating graphic content. Although, graphic primitives provided for such APIs are quite simple (points, lines, triangles, squares), and you need a lot of work to get some result. Also, since new graphic cards have become an unified architecture without fixed function, for whatever effect we would like to create we must create a shader, which is not an easy thing for complex effects. Here is where game engines appear. Game engines provide a higher level of abstraction, being capable of working with models, not triangles, effects,… providing a friendly interface to work with. Working with a game engine is quite more simple than with the API itself, and most game programmers work with it to create their content.

Now we are going to describe more accuracy each level.

**3D Game engine**

A 3D Game engine is the highest level view which provides basically all the services a game might require to render a game. 3D game engines deals with meshes, bones, effects, textures and so forth. It offers a simple interface so that the user of the engines does little more than choose what object to render with which materials and how. To do so, a modern 3D engine uses an API that will communicate, through a driver, to the hardware.
3D game engine are normally included inside Game engines which provides all services a game might require to run, such as sound modules, network, I/O modules.

The way that 3D game engines are design are to be simple, fast, efficient and elegant. A 3D engine’s task is only to render the world to the screen, and it might interact with the disk I/O module to load data when is needed. To be more accurate, it should be said that only the player’s interest must be displayed. It obviously means that the 3D engine does only have to render a subset of the complete game world, which is the part visible in the viewport. So, one of the tasks of a 3D engine, is to find as quickly as possible the visible subset of the game world. To achieve that, the world is divided into areas, which store the objects they fully enclose. Then the engine will find the areas visible from the camera point of view, and know which objects to render. This process is often referred to as culling.

Rendering is another task of the 3D engine. Once it has found the smallest subset of objects to be drawn, it must render them as quickly as possible. The independent hardware vendors (IHVs), AMD, NVIDIA, and others, have published many documents about the best methods to render a scene quickly, out of which two major points always come up: minimising your state changes, and batching your geometry.

There’s yet one other task of the 3D engine, and that’s to animate characters, which is mostly done today through skinning. Skinning is a process in which the bone of a skeleton are hierarchically transformed (so as to have a child bone move with its parent), and their resulting position and orientation used to place the vertics (the skin) of the characters at the right place. This process is often perform during frame updating.

Those three steps are performed in the following order: updating, culling, rendering and to be googd, a 3D engine must perform all those steps efficiently, and to do that involves using a 3D API properly.

Normally a game engine performs all the graphical work, but some engines only do one thing, but they do it more convincingly or more efficiently than general purpose engines. For example SpeedTree was used to render realistic trees and vegetation in the role play game The Elder Scrools IV: Oblivion.

Some examples of game engines being used for famous games are Cry Engine which is used in Crysis and Crysis Warhead in its second version. Another one is Source Engine, developed by Valve which is mostly used by valve games such as HalfLife.

**Graphic API**

Graphic API is lower level of abstraction. In this level API interface deals with Vertex Buffers, Index Buffers, Shaders, Textures, RenderTargets.
There are two main 3D API available today: Direct3D and openGL. Both provide an interface to the same underlying hardware, with the differences being the quality and simplicity of the interface, and the implementations (the drivers), rather than the feature set.

On one hand you have Direct3D, pushed by Microsoft, with a rather nice interface (in its 9th and 10th versions), but suffering from a severe draw call issue. Draw calls on certain Direct3D platforms force a kernel context switch, which an ultimate performance has cost. The other downside is that the API tends to change a lot from one version to the other, which isn’t too nice since it means rewriting a lot of code to take advantage of any new version in any existing engine. D3D9 to D3D10 transition is a good example of that problem. It’s available on Windows and Xbox 360.

On the other hand you have openGL, lead by the Khronos Group (and formerly by the architecture review board (ARB)), and exists in different flavors such as the ES (Embedded Systems) version, or the standard openGL for workstations. openGL 2.1 has many functions, and many ways of doing the same thing, making its implementation difficult and the engine write’s task uneasy, as there is a need to look for the optimal path, which evolves with time and new extensions. Or new hardware, since extensions are meant to make hardware features available in the API, without breaking the existing interface. Hopefully, openGL 3.0 has a completely new streamlined interface, sometimes referred to as “Learn & Mean”. It’s available on MacOS X, Windows, PS3 and Wii.

There’s the option of deciding to follow one of the API and use its strengths in the engine. While it allows to take advantage of specific features provided by chosen API, it also means restricting the system the engine will run on, and so the number of potential users of the engine. A more interesting approach is to choose neither of them, and to write an abstraction layer which will hide all API specific code inside a module, making the engine API agnostic. With such a layer, the engine will be able to use the best API for a given system, to ensure high performance. The drawback of having of having an abstract renderer interface is that it must target least common denominator of the APIs it’ll be hiding, or the engine will need some tweaks to target some platforms. Still, since the code is nicely encapsulated, changes, even engine bread, will be much easier to deal with.

Both of them allow us, instead specifying individual vertex data in immediate mode (between glBegin() and glEnd() pars), to store vertex data in a set of arrays including vertex coordinates, normal, texture coordinates and color information, and then draw calls can be made by referencing those arrays. Draw calls that uses those arrays are glDrawArrays(), glDrawElements and glDrawRangeElements. Vertex data is stored in raw buffers which can multiple vertex data. Once the programmer would like to active or deactivate one of the different vertex data it only has to execute glEnableClientStat() or glDisableClientStat() ir order to active or deactivate the different types of arrays. Moreover, when this is done, we need to set the buffer we are going to use. Setting the buffers can be done using the calls gl***Pointer where *** can
be Vertex, normal, color, index, texcoord or edgeFlag. Vertex Arrays and VBO work in a different manner. With Vertex Arrays glGetPointer functions are used without nothing more. This call has as parameters the type of data is going to be used, the stride between elements and the pointer to the raw data. Apart from data nothing else must be set up, only as mane arrays as needed. When VBO are used more thing are needed to do. First raw arrays must be introduced inside Buffer objects using to do this the glBufferDataARB functions. Parameters for this call are basically those to characterize the buffer content, such as the buffer size, data pointer and the usage. Every buffer we would like to use must be created using this function. When all buffers have been created we can change between them using the glBindBufferARB call. This function alls us to set one buffer, remember that OpenGL works as a state machine. Then, as with Vertex Array has been done, glGetPointer calls must be used to set the appropriate buffer to each characteristic, but this time the third parameter which is the pointer to the data won't work the same way as before. This time to set the buffer that is going to be used will be done choosing the last one settled using the glBindBufferARB call, and the pointer parameter will be the offset inside this buffer where data can be found.

Driver

A driver is the most hardware dependant piece of low-level software. Writing a driver requires an in-depth understanding of the hardware we are going to make the driver for as also a good knowledge of the OS on which we are going to build the driver. As you probably noticed, driver is a high OS dependant code.

GPU drivers typically communicate with the GPU card through the computer bus that in the case of GPU is a AGP bus for the old fashion computers or the new PCI Express bus with all versions of different speeds for the new ones.

GPU driver main functions are making transparent to the above levels the specific characteristics of the underlying hardware. Many times computer architectures design hardware having some weaknesses that may cause a great overhead in case the hardware utilization is not performed properly. An example of this may be when a given functional unit of the hardware is terrible slow and in order to maintain the throughput of the system the drivers tries to change all the operations regarding this unit to other operations that don't affect so much the system performance.

Another important task for the GPU driver is the GPU memory management. Driver must allocate and deallocate memory in a proper manner in order to reduce the bus overhead. Also data placement is important in order to avoid in and out operations from memory.

GPU (Graphic Processing Unit)

GPU is a highly dedicated piece of hardware which is design as a specific purpose processor implementing a specific 3D rendering algorithm.
This section explains the basic GPU pipeline. Explanation is not very in depth and will only talk about basic GPU architecture. Due to rapid evolution GPU architecture have we are going to take a glimpse about how this evolution take place.

**Approach to existing architectures**

New GPU architecture can be divided in two different parts: the fixed and the programmable part. The programmable part consists of some small processors called stream processor being capable of executing the shader programs we describe in the previous section. On the other hand we have the fixed part which has some functionalities that cannot be expanded only some setting about how they work but nothing about adding new functionalities.

Next picture maps each API stage to the correspondent in the GPU one.

Streamer is the first stage a GPU has and is the responsible for reading all the vertex data provided by the API. This data is all the vertices that are going to be render this batch as well as all the information they have associated. Vertices can be selected using the index buffer in case it is an indexed draw call. Once all the vertices have been read next operation is vertex shading. Vertex shading is performed over each vertex read by the streamer and it is done using the stream processors. After vertex shading next step is join the vertices into the primitive chosen by the programmer. Beyond this point we are working with primitives, usually a triangle. Next step is executing over each primitive the geometry shader, as we had seen before, for each geometry shader is executed over each primitive and this is the first stage where primitives can be discarded.

After all the work over the primitives is done is time to transform primitives into fragments. This operation is performed inside the rasterizer. Rasterizer has two main steps, initially compute all vectors that define the primitive edges and once this is done use them to generate fragments from the primitive we are working with, this process is done in fragment generation. **EXPLICAR A GRANS TRETS COM ES PRODUEIX LA RASTERIZACIÓ** After rasterization our working object are fragments. The following stages has as main goal discard as many fragments as they can in order to avoid extra computing costs. It is difficult to know if a fragment will become a pixel because the only known information comes from all the previous pixels that has been written on the frame buffer. In fact, a fragment that has been written does not mean that it becomes a pixel because another fragment may be written over this before the draw call has finished. So discarding information can only be obtained from the fragments that are on the frame buffer, and all the information will be obtained from the Z value they have. Even so, as we seen when we were studying the API pipeline other test can be performed to discard more fragment, but those tests are designed by the programmer.

First test performed over the fragments is early Z test. This test performs a fast comparison between the Z of the fragment we are evaluating and the fragment written in the frame buffer.
As the test is performed in early stages other fragments can be on fly inside the pipeline so this test only guaranties as that if the result is to discard it is correct, otherwise no final conclusion can be reached. Next test is Hierarchical Z, this test is also done using the Z value as main information.

After these first tests are done, every fragment is processed in a fragment shader. Before this, fragment attribute interpolation must be done because rasterization haven’t done it yet. As we previously explained fragment shader decide which will be the final color the fragment will have. As geometry shader, fragment shader can also discard fragments. Fragment shader can perform texture access throw Texture Unit which will return the color that is map on the coordinates given by the shader. Texture Unit is responsible of performing all the texture access needed to obtain the final color based on the filter type the programmer has chosen.

Once fragment shader is done is time to do other test in order to discard more fragments. Next testes have been explained when we have explained the API pipeline so they will not be explained another time.

Discarding fragments is not so easy to do in all situations. When we are working with alpha blending active, fragments written in the frame buffer can have some degree of transparency, so even a fragment is rear this fragment is need not to discard it because as the fragment written has some transparency we can see the fragment rear it. So in this situations, early Z and Hierarchical Z are disabled to avoid errors discarding fragments.

Once all tests have been done, fragments are ready to be written in the fragment buffer and this is done in the blending stage. This stage is responsible of writing the fragment on the frame buffer and performs any operation between the fragment currently in the frame buffer and the new one in order to obtain the final color. Writing the fragment on the frame buffer also implies a modification on the z buffer in order to have it up to date.
3. **ATTILA**

The previous chapters explained the basic graphic theory required to understand the whole project. This chapter continues explaining the working environment used in this project which includes the entire Attila framework. This framework is really big, as a result of many years working on it, and this section only talk about essential things needed to understand this project.

**Workload generation**

Attila is a cycle-accurate GPU simulator which its main objective is test new architecture proposals and evaluate them. In order to evaluate the proposals, the workload used has to be as realistic as possible to be sure that if the architecture proposal is implemented on hardware and used in real environments it will behave as in the simulator. The best way to succeed in this task is using real workload to feed the simulator. Nowadays the most representative workload for GPUs are PC games, although console games can also be used but the problem with them is that they are played in closed environments for which you have to pay some royalties to obtain the programming kit.

The workload is not executed directly on the simulator due to some drawbacks. First of all, the simulator is several times much slower than a real GPU, each frame could take about 30 minutes\(^1\) to become simulate, so it is not possible to play with game on the simulator. Another important inconvenient that playing directly a game on the simulator has is that every time the game is played the API calls issued are different as it is not possible to play the game exactly the same way all the times, so the workload is different every time the game is played.

\(^1\) Time needed to render a game depends on the capabilities and resolution the game uses. This time is an example from a simulated frame from the Doom 3 game (openGL) with a resolution of 320x240.
This represents an important problem as there is no way to compare different architectures with the same workload, something that is critical when an architecture proposal is evaluated.

To solve these two important problems, instead of playing the games on the simulator, they are played on the real hardware and, using an interceptor, the API calls the game issues are recorded into a file called trace. This file contains all the API calls a game issued as well as all the information needed to execute these calls, such as buffer contents, etc. Is true that games perform many other operations such as collisions, IA computing, physics among others, but these operations are computed on the CPU, and this information is not required to evaluate the graphic hardware.

Working this way solves all the problems; from one side the workload is always the same because what is used to reproduce it is the trace content. From the other side no matter how long it takes to be simulated as the user don’t have to control the game.

The way API calls can be intercepted is not so simple. DirectX and openGL are exported as dynamic libraries so the best way to capture all the API calls games issue is to interpose between the application it is desired to trace and the graphic API it is desired to capture another dynamic library that intercepts all the calls, this layer is called interceptor. So the interceptor acts as a fake API library and games make their calls on it, then the interceptor collects all the information necessary to later reproduce the behavior and finally returns the call to the original API.

GLInterceptor is the tool developed by the group to perform this task and record traces of openGL games while Microsoft Pix Tool, pushed by Microsoft, is used for DirectX games. Both applications work in a similar way and as a result from their execution, one or several files are obtained which are the trace files. Information about how the GLInterceptor was designed and built can be found in (González, 2004)

Once the trace is obtained the game is not need anymore because all the information needed is contained in the trace. Next step is being able to reproduce the trace file.

This schema shows the basic architecture used to capture and play the workload on the simulator.

---

1 Now IA computing and physics can be computed using the GPU, but this is not considered graphic computing
Player

Traces can be played in two different environments, on a real GPU or on the simulator. Playing the trace on a real GPU is useful because it is possible to compare the behavior of the trace against the real game and check that the trace is properly recorded. This player can also output batches and frames. On the other hand, playing the trace on the simulator is necessary to feed the simulator. The simulator also provides as output frames and batches from the trace. Then it is possible to compare these frames and batches against the player ones and check that the results are more or less the same.

Before going on is necessary to explain 2 new concepts introduced: frame and batch. A frame is when the back buffer is exchanged with the frame buffer and a new image appears on the screen. This is done using an API call and to be able to see movement it is necessary that this happens at least 24 times per second. A batch takes place when a draw call is issued. This batch includes the entire context necessary to render the geometry that the draw call contains. Then, each frame is composed as a sequence of batches, and the number of these batches in a frame depends on how complex it is and how many different states it needs to be drawn.

Figure 20: Diagram that explains how the workload is obtained from real games (left side), how it is reproduced on real hardware (center side) and how it is reproduced on the Attila framework (right side).
The way the player works in both scenarios is more or less the same, first task is to read the trace file. Reading the openGL trace file format is easy as the format was created by the group. The problem comes with Pix which trace format is coded by Microsoft so it isn’t so easy to read the trace file. Information about the Pix trace format and the DirectX player can be found in (Solis, 2007)

From the trace readed, a sequence of commands is obtained. These commands must be sent to the appropriate stack, to the IHV API versions in case it has to be played on a real GPU or to the emulation stack in case it feed the simulator. The way this is mainly done is using tables that gold pointers to the appropriate calls. This table is set with the appropriate pointers (IHV driver functions or the emulation driver functions) to each of the implemented functions.

**API**

Attila supports two different APIs: openGL and D3D9. This section doesn’t explain them in depth, only a few details are given. If the reader wants more information about how they are built, good references are (González, 2004) and (Solis, 2007)

As games only use a small subset of the available functions an API has, it is illogic to implement all of them. The most logic approach is to implement the function call needed on demand, avoiding implementing things that are not going to be used.

Before starting this project, the openGL API implemented in Attila supported a large subset of openGL functions and some of the most important API features from openGL specs from 1.4 to 2.0. With the implemented capabilities the games supported were: Doom 3, Quake 4, Prey, Chronicles of Riddick and Unreal Tournament 2004. To play them about 200 API calls were supported. ARB shader language was supported and translated into Attila Byte Code. A great range of texture formats were supported including support for S3 Texture Compression (S3TC) modes (DXT1, DXT3, DXT5). The texture filters available are nearest, bilinear, trilinear and 16X Anisotropic. OpenGL also support different ways to define vertices. glBegin()/glEnd() structure, Vertex Arrays and Vertex Buffer Objects (VBO) are supported and fully functional. Fixed function is supported using the fixed function shader code generator proposed in (Roca Monfort, 2005)

The D3D9 API supports all the available D3D9 draw calls as well as managing vertex buffers, index buffers and user pointer streaming mode. Vertex and pixel shaders are supported up to version 3_0. 2D, 3D and volume texture are supported using the most common texture formats as well as the S3TC compression formats. Using render targets and Z, alpha and stencil tests are also available. With all this capabilities the Half Life 2 game is supported.
Driver

The driver is the lowest piece of software and it is just above the simulator. As it is a low level code, its main functions are abstract some basic resources avoiding that the API programmer have to deal with them. In Attila, the driver only takes care about the following things:

The GPU has its own memory space. Handle this memory is a difficult task as it is possible to overwrite it if it is not managed appropriately. To avoid this, the driver's task is manage this memory and provide the upper levels some operations to request, release and modify this memory. The driver don't expose memory addresses to the upper levels, instate of this, it provides memory descriptors (MD) which are nothing more than a number that the driver maps to the appropriate memory region. The GPU memory stores all the resources needed to execute a single batch; these resources are shaders, textures, render targets, buffers...

GPU's are connected with some sort of interface to the CPU. Attila simulator is connected using an AGP interface. Then the way the CPU and the GPU are communicated is issuing AGP transactions. A transaction is a low level communication unit that is not desired to be exposed to the user because is something that depends on the interface used by the GPU. It is not logical that it is need an API version for each possible interface. Also GPU manufactures don't want that their users know too many things about their architectures. These transactions are used to write the GPU registers.

The driver also has a shader cache. The reason why the driver has a shader cache is because of the Attila simulator. Shaders, as other resources, are saved in the GPU memory, the problem is that the simulator can only execute the shaders if they are in some special memory region, which size is limited. Then not all the shader can be saved in this memory area, only some of them. Then every time a shader is wanted to be run, it is need to check if the shader is in this memory space, if not, it must be copied from the memory region where it is saved to this region. The problem is that this operation takes too much time, more if the workload changes many times the shaders it uses. To avoid performing many times this operation a shader cache was introduces in the driver.

The Attila shader units execute the Attila byte code. Also it is available the Attila assembler code, that is a high level code that can be used to program shaders. The driver offers a compiler from this assembler code to the Attila byte code. The driver does another task. Texture units from the Attila simulator read the textures using the Morton layout to improve the spatial locality. Typically textures are defined row by row, so before they can be used it is necessary to transform them to the Morton layout. The driver provides one function that given the raw texture data, change its layout to Morton.

More information about how the driver is designed can be found in (González, 2004)
Simulator

This last section explains the basic architecture of the simulator. The aim of the section is explain the basic characteristics of the simulator without too much detail. More information can be found in (Moya, González, Roca, Fernández, & Espasa, 2006)

The following schema shows the basic elements of the simulator architecture:

The GPU memory is found in the right side of the schema and it contains all the resources such as shaders, textures, buffers and render targets. This memory is accessed using a memory controller which simulates a GDDR memory interface.

The left side of the schema shows the pipeline.

The first stage is the streamer which internally is divided in different stages, the index and vertex fetching, as well as a vertex cache and some available stream units.
First, index fetch unit reads the indices from memory using the stream where the indices are saved. The indices are transferred to the vertex fetch unit where they are used to know which are the vertices it has to read. Vertex fetch reads the vertices from the vertex cache. If the desired vertex is not inside the vertex cache, a miss occurs and the vertex cache has to request the vertex attributes (position, color, texture coordinates…) to the memory controller using another time the stream units. When the information is received, this vertex is send to one of the shader units to do the vertex shading. There are more than one shader unit, so to schedule to which shader unit one vertex has to go there is a scheduler. Scheduler main task is to distribute work among all the available resources. This work can be vertices or fragments that are waiting for one of the available resources that can be the shader units, the interpolator, or the ROPs. In this case the vertices have to do the vertex shading, so they require a shader unit. This shader unit executes the selected vertex shader program that must be inside a special memory region.

Once vertex shading is finished, the shaded vertex is returned to the vertex cache and is saved there and returned to the vertex fetch. Then, the vertex cache what saves is shaded vertices.

Is important to notice that Attila has an unified shader architecture, which means that shader units can execute whatever shader type. There are other architectures which has specific shaders for vertex and fragment shading.

Vertices that output the streamer enter inside the primitive assembly. This stage joins some vertices in a primitive. The way how these vertices are arranged depend on the primitive type. The simulator supports the following primitives: triangle, triangle strip, quad and quad strip. So, the outputs of this stage are primitives.

Next stages are culling and clipping that as they name say do the culling and clipping tests to discard geometry. Next to these stages is the rasterizer. This stage generates the fragments from the input geometry. This stage is also divided in two smaller stages, the triangle setup and the fragment generation. The triangle setup stage calculates the triangle edge equations and a depth interpolation equation while Fragment Generator stage traverses the whole triangle generating tiles of fragments.

Next stages do the early Z and Hierarchical Z tests to discard fragments.

Fragments that arrive here are sent to the ROPz throw the scheduler. ROPz includes the scissor test, stencil test and Z test. These tests accesses the memory throw the memory controller.
Fragment that output the ROPz are sent to the Interpolator that interpolates the fragment attributes using one of the available interpolation types are linear, centroid or no perspective.

These fragments are sent to the shader units to do the fragment shading. The fragment program can do texture access throw the texture unit, which take cares of the address mode and do the appropriate filter to the texture access. The texture unit output is a color sampled from the texture. Texture unit can access to texture resources as well as to render targets.

Textures units read the textures using the Morton ordination, so the textures that are stored inside the memory must be saved using this layout. On the other hand Render Target resources use a different layout, called render target layout, and this may be taken into consideration when the user is working with them.

Finally fragments are sent to the ROPcolor where the blending stage take place on the render target set.

The Attila simulator supports multiple render targets (MRT) what mean that it is possible to set multiple render targets at the time. The maximum number of render targets set depends on how the simulator is configured, but the main render target is the render target 0. Then, when multiple render target are set the blending stage take place on all of them, and each one can be configured in a different way.

The entire GPU pipeline is controlled by a Command Processor which also receives the AGP transactions described in the driver.

Many of the capabilities the Attila simulator provides have not been explained here and are explained in the following chapters.
Previous chapters introduced the graphic theory and the environment used in this project. This chapter focuses on analyzing which are the main challenges of this project, which are the problems it tries to solve and which are the possible solutions. The chapter is divided into 3 different parts. First it analyzes which is the problem it tries to solve, then continues explaining which are the possible solutions to solve the problem and it finishes describing which is the final solution.

**Initial problem**

Let’s take a moment to review how Attila stack is built.

![Current Attila framework](image)

**Figure 22: Current Attila framework**
At the top level there is one player for each API which its main function is reading the trace files where the workload is saved and interprets them issuing the appropriate function calls. Under each player there is each graphic API supported by Attila, right now openGL and D3D9. Just between the APIs and the simulator lays a thin driver which main task is manage the memory and the AGP transactions. Finally the Attila simulator does the low level work simulating how a real GPU behaves and rendering frames from the provided workload.

The current architecture has several problems that must be analyzed. Let's make an overview of them.

At first glance, the main problem is that each API supported is implemented on the driver and is entirely independent of each other. This fact has many consequences.

Being each API entirely independent means that they don’t share any common functionality and this is not true. The hardware underlying the APIs is the same, so the way it has to be configured is quite alike in both cases. Then, it seems obvious that these functionalities must be shared, and not doing this has some consequences.

One important drawback is that this model doesn’t reuse the code, which is one of the key principles of the computer studies. As the code is not reused, more people are needed to program and maintain each API, something that is not possible if the human resources are restricted.

But this is a small problem between others bigger. Another problem appears when changes are made in underlying levels, such as the driver or the simulator. Typically when some changes are applied to the simulator is because some bugs are fixed, which normally don’t cause changes on upper levels, or because new functionalities are added. Always adding more features is good, because probably more games may be supported, but in this case this is a problem, because this new capability has to be added to each API, which mean more work and a waste of time.

But this is not the only problem. Probably each API is programmed and maintained by a different group of people, people which may use a different philosophy and structures to program the API. This causes that different bugs appear on each API, which probably don’t have any relation between them. Another time more time have to be invested to solve all this problems.

This stack is used in a research group where occasionally some of its members would like to optimize how the API uses the underlying hardware. With the current framework, this member would need to study how each API is design to add the optimization to it.
These are the main disadvantages the current design has, but let’s thought it over. Now, there are only 2 different APIs, openGL and D3D9, but new ones have been released, D3D10 and D3D11, and many other exist such as openGL/ES. Now imagine that three more APIs are added to the framework. Then Attila will support up to five different APIs. First problem is that every new API have to be programmed starting from scratch. Also all the previous drawbacks described will be multiplied by five.

By the time, current framework design has worked, but it is obvious that its current design is the bottleneck of the system because it requires too much work to add new APIs or new capabilities to the underlying system. So the aim of this project is suggest a new structure that solves these problems.

**Possible solutions**

Solving this problem is not as easy as it may seem. Programming the openGL and the D3D9 APIs took so much time, so it is desirable not throwing away all this work and reuse as much as possible.

In order to find the appropriate solution, the first step is to define which are the main goals the solution should have.

One of the most important problems the current architecture has is that all the APIs have many similarities in the way they manage the underlying hardware but they don’t share their code; and as they don’t share code, each new added API should be started from scratch and must do the low level hardware management.

The best solution to this problem is to create a new layer that collects all the common functionalities all the APIs have. Then, each API only implements its specific functionalities and uses those provided by the shared layer. This solution solves the problems of not reusing the code, as well as reduces the amount of work needed to start a new API, as it can use the shared functionalities.

Next step is to analyze which are these functionalities that can be shared among all the APIs. It is important to understand that there are two different views about how the resources and the state of the GPU are managed; from one side is how it is managed from the point of view of the hardware, and from the other side how they are managed from the point of view of the API. Understand this is very important. All the APIs manage the hardware the same way as there is only one hardware and it provides the same interface to the upper levels. Otherwise, there is more than one API and each one of them is manage in a different way.
It's quite obvious that the way each API manage the resources cannot be modified because it is defined by the API definition and it is logical that the way how the resources and the state of the GPU are managed can be gathered into a shared layer.

Then this first decision remove all the low level code to manage the hardware that each API has and put it together in a new layer. This new layer provides the resource and state management of the hardware and exposes a set of functions that the upper levels can use to manage the hardware. In fact this new layer can be thought as a new API, because it provides functions to manage the underlying hardware but using its own interface. Then each specific API has to translate its specific API specification to the shared API one.

Next important feature the solution should provide is a way other people can easily incorporate their API optimizations. These optimizations are almost always hardware dependent, so the best place where they can be is in the shared layer, because is where the lowest API management code is and also because is shared among all the specific APIs which fulfills the requirement of only having to implement them once.

Typical optimizations that can be here are those related with the shader programs and data organization. The reason is quite obvious looking how GPU architectures have advanced to completely programmable architectures where the most important thing is how data is spread in memory and how many instructions the shader code has.

Finally the new architecture also should provide an easy way to debug the code and decrease the time needed to implement new features. This is an important part of this project as current API implementations don't have the appropriate debugging tools. Debugging tools can be placed everywhere, each specific API or in the common API, but it is obvious that many of them must be inside the common API.

Apart from the described characteristics, there is another one that is more connected with how new GPUs work. As previously seen in page 37, transform and lighting stages have been removed from current GPU architectures being substitute by programmable units called shaders. This also took place in the Attila architecture and the problem is that many current APIs continue having fixed function support. As the simulator doesn't provide support for fixed function, the only way to execute these stages is emulating them. This is done generating an equivalent shader code that behaves the same way as the configured transform and lighting stages.

To generate the shader code it is need to provide a fixed function to shader code generator, which generates equivalent code to emulate the fixed function stages. All APIs fully or partially supporting fixed function stages must implement this shader code generator. Implementing one shader generator for each API that needs it have the same problems as those found implementing each API completely isolated, so the solution is another time implement a
common shader code generator that can be shared by all the API that need it. As this code is shared by many of the API it seems obvious that it can be inside the shared common API. But this statement is totally wrong because this unit will be only used by some APIs, and also it is legacy code to support old capabilities, this characteristics don’t agree with the common API definition which contains common code to manage the state and the resources of the underlying hardware. So it will be implemented in an isolated structure that whatever API can use. Implementing this fixed function code generator is not an easy task and requires a deep knowledge about shader programming. This part has not been developed in this project and it has been provide by another member of the group. Its specification and implementation is based in the following publication (Roca Monfort, 2005)

Another important topic to talk about is the fact that the simulator shader units only execute Attila Byte Code. This is a special shader format designed for the simulator, so native shader languages used by the APIs are not compatible with this language. Then another important tool needed is a shader compiler that compiles from a given shader language to the Attila Byte Code. It is need one shader compiler for each available shader language, and as it is possible that some APIs share their available shader language these compilers can be shared. The solution is to implement these compilers in an isolated structure that can be used by all the specific APIs they need them.

Project proposal

Across all the topics discussed in the previous section some conclusions can be reached. The most important is that the API layer has been divided in two different layers. One of them is the common layer which manages the state and the resources at low level. This layer can be seen as an API that provides an abstraction layer for the Attila architecture giving a friendly high-level interface, suitable for building specific APIs on top. This layer is coined as Attila Common Library (from now on ACD). The ACD will manage the state and all the resources provided by the simulator, so any specific API won’t have to manage low level configurations. The ACD also provides better debugging tools as well as a place where the optimizations can be placed.

On top of the ACD each different specific API is implemented. Each specific API can be seen as a layer that translates from the specific API interface definition to the ACD interface one.

Apart from those two different layer, there are also available two more elements; a fixed function shader generator is used to generate shader code from a fixed function definition. Also, some shader compilers are used to compile each specific shader language into the Attila Byte Code.

From the previous description the following schema can be obtained:
The players, the driver and the simulator haven’t changed from the previous design, but now between the player and the driver new levels have appeared. Players continue issuing calls on each API, but now these APIs have been reduced and all their common functionalities moved to the ACD layer that is down the specific APIs. The ACD manage the underlying hardware throw the driver. Finally the fixed function shader code generator and all the shader compilers have been introduced into a new unit called Attila Common Driver eXtensions (ACDX). The ACDX can be accessed by whatever specific API that needs some of the functionalities it offers.

The main objective of this project is to design the ACD layer since it is essential to continue developing the Attila project. The problem is that it is too difficult to evaluate if the ACD works well because there aren’t workload with the ACD interface, so the best way to solve this problem is to design on of the specific APIs above it. As openGL driver is more developed, we decided that the best shot is building the AOGL on top of the ACD. Building the AOGL allows to check if the ACD fulfill with which it is designed and also test that the ACD works properly using real openGL workload.

The ACDX module is also need to develop the project because it is need to compile the ARB code into the Attila Byte Code and also because some games use some fixed function characteristics. Luckily this part is done by Jordi Roca, another member of the Attila group.
The previous chapter explained the current Attila framework and which are its main disadvantages. To solve these problems a new architecture is proposed. This architecture is designed with two different levels. The ACD is the lower level that manages the resources and the GPU state and exports an interface that can be used by other specific APIs. On top of this layer each specific API translate the specific API interface to the ACD one.

The next three chapters are focused on explaining how the ACD is built. First chapter is focused on the ACD interface. It is addressed to those people who are going to use the ACD interface to build a specific API on top of it, and it explains how the ACD interface is designed and which are the functionalities it offers. Second chapter explains how the ACD is internally designed and it is addressed to those people that need to understand the whole ACD, for example to know where to add a specific optimization. Finally, last chapter explains how some parts of the ACD are implemented. These parts are the most complex ones and it is interesting to explain them in case it is necessary to modify them in the future.

This first chapter analyzes which are the main requirements an interface should have, and explains which is the ACD interface design.

**Interface characteristics**

An interface is nothing more than a set of operations provided for a device that expose their functionalities and let an outside user use them without any knowledge about how they are implemented internally.

Designing an interface is not an easy task. The way it is designed affects the people who use it as well as the people who program it. Choose a bad interface can cause that it becomes more
difficult to use it by other people or more difficult to implement the interface. People who designs them must have a deep knowledge of the underlying device an also how this device is going to be used by other people.

There are some important characteristics an interface should have and it is necessary to study them before continue.

May be, the most important characteristic an interface should have is that it have to be complete which means that it must provide all the functionalities the underlying software have, and it must be done in such a way that allow the user to obtain the maxim benefit from those functionalities. It is necessary to take into consideration that in the future more functionalities may be added, so it is also necessary to provide an appropriate design which allows that these future modifications can take place without having too much troubles.

An interface must also provide a clean interface, where operations with similar functionalities are grouped together. Methods and parameters names have to be chosen appropriately as they must point out what they mean without having to read the specification every time.

Moreover, the same functionality should not be duplicated in different operations because it might confuse the user about which operation it might be used. Also, an excess of worthless operation should be avoided as it increases the size of the interface making it more difficult to use. An interface should make everything as easy as possible.

Another important characteristic is that the interface should be stable and by stable we mean that new modifications done on the interface should minimize the consequences on previous implemented methods. This is important because if previous methods are modified very often, the people who use them should modify their code to continue using the interface.

Debugging is also a task that the interface should help with Report the errors to the user in a clear way might help him to understand what have happened and solve it. Also, provide good debugging tools may help a lot.

Before start thinking how the ACD interface has to be design it can be interesting to analyze other graphic interfaces and which have been their main errors. As Attila framework supports openGL and D3D9, so they are the appropriate ones to be analyzed.

OpenGL problems appear due to how complex its interface is, having multiple paths to do the same thing. This is a problem because in case it is desired to optimize the code it is difficult to know which path provides the lowest delay. On the other hand, DirectX provides a clean interface but have another important problem, and it is how different is each version from the previous one. Then, if a new DirectX version is released, and your application, which is
programmed using the previous version, would like to take benefit of the new version capabilities, it is required to practically reprogram all the application because the changes between each version are so big that they are not compatible. This problem has been taken into consideration and new D3D10 and D3D11 have a more stable interface.

Apart from the previous characteristics, the ACD interface should have another ones derived from the ACD characteristics described in the previous chapter. The ACD interface is used by other specific APIs which use the services it offers. To accommodate all this variety of APIs and reduce the work to do in each specific API the best way is provide an interface as close to the specifics APIs ones as possible. The problem with this approach is that if a new API, with a very different interface, is added it can be much more difficult to adapt. To solve this problem the ACD interface also should try to provide an interface as generic as possible to avoid this problems. Try to fulfill both requirements is another task of the ACD.

**How the ACD interface is designed**

Designing the interface for the ACD is not an easy task. Many things have to be considered, and without any experience designing them, it is even more difficult. If the people who are designing the openGL and DirectX have errors, imagine a person who never designed any one. Because of this, the ACD take some ideas from other APIs. After analyzing all the available APIs, the conclusion reached is that the best interface is provided by D3D10\(^1\), because is which better fulfill the API characteristics described in the previous sections. Then to build the ACD interfaces, the D3D10 served as inspiration.

Designing the interface has two different parts, the interface necessary to manage the resource and the interface necessary to manage the state. This is the same division used when the ACD was described in chapter 4.

This first schema explains how is built the interface to manage the resources:

\(^1\) When the project started the D3D11 haven’t been released yet.
The interface to manage the resources is totally hierarchical. The first element is the Resource abstract object that contains all the information that all the available resources have in common. From the Resource each specific resource inherits, adding its specific characteristics. The available resources are the ACDTexture, the ACDRenderTarget, the ACDSHaderProgram and the ACDBuffer. It is possible that a single resource type have multiple subtypes, for example the ACDTexture. In this case each ACDTexture subtype inherits from the main texture object which has the common texture characteristics and is an abstract class. The schema is simple, an in case more resources are added they only need to inherit from the main Resource object, and if it has more subtypes, they must inherit from the main type.

Finally point out that the ACDRenderTarget is created from an ACDTexture object, that why there is a relation between these two object.

Next important block is the state management. The first idea was to create one structure for each functional unit the simulator has. This is a good architecture proposal as emulates the hardware design. Then, each different structure is responsible for synchronizing its units. The problem is that there are some capabilities that don’t belong to any functional unit, such as the draw calls, clear operations, etc., so it is need a place where to place them. Also there are too many functional units, and many of them only have one or two parameters to configure the functional unit. The fact that there are too much structures also makes it more difficult to configure the ACD, as the person who uses it must take care that all the units are synchronized with the GPU. To solve this problem two different solutions where proposed. First one is add a new structure called ACDDevice which contains all the operations that don’t belong to any specific functional unit. The second solution is, instead of building one structure for each functional unit, only build them for the bigger ones, and for the others, which don’t have many operations, add them to the ACDDevice.

Then, from the previous description the following class diagram is obtained.
Figure 25: Class diagram of the state management. The relation between the ACDDevice and the ACDStream or the ACDSampler is 1 to * where the * value is the maximum number of streams or samples the underlying hardware has been configured with.

From the above class diagram we can see that there are some structures used to control some parts of the GPU state. These structures are the ACDStream, the ACDRasterizationStage, the ACDSampler, the ACDZStencilStage and the ACDBlendingStage. For the ACDStream and the ACDSampler, there are as many of this objects as stream and sampler units has the underlying hardware has been configured with.

Apart from these structures there is one more, the ACDDevice, that controls some other parts of the GPU state and manage all the structures previously described.

The following schema shows which are all the functions the ACDDevice controls and also maps the resources described before with the structure they are used.
Figure 26: Stage management design. Includes how the resources

On the right side of the schema there is the ACDDevice, which is the unit responsible for managing the whole state. Inside it there are some boxes that represent the different units the graphic pipeline have. All of them have been described in the chapter 2. As previously seen for each of the stages one structure is created, but some of them are implemented inside the ACDDevice. Then the red boxes represent graphic stages mapped inside the ACDDevice while the green boxes are stages that are implemented in a separate structure. Finally the orange boxes are the available resources and they are placed next to the unit that uses them.

Next sections describe more in depth how each of these structures is built and used.
Resource management

This section explains the available resources and how they can be configured. It is important to notice that none of the resources can be instanced by the specific API, so if the user would like to create one of them, it must be created using the available functions in the ACDDevice. This topic is discussed when the ACDDevice is explained.

Resource

The resource is the basic class from which other available resources inherit, so information stored here is those that characterize all the possible available resources. It is important to remember that there are two possible memory locations where data can be stored, the GPU memory and system memory. The GPU memory provides low latency when it is accessed from the GPU but high latencies when data must be transferred there from the system memory as it must be done through the AGP bus. Otherwise, system memory provides high latencies when it is accessed by the GPU but low latencies when it is accessed from the CPU. Where to place the data is a difficult task as it depends on how it is used by the programmer. This information is only known by the user as he is using it, so he must be who tells where to place the data.

If data is created and then only used, it must be placed in the GPU, as there are only a few write operations which take a little bit more because they have to be done through the AGP interface but read operations take much less as the information is stored in GPU memory and accesses to it are quicker. Otherwise, if data is going to be modified very often it is better to place it in the system memory as modifications can be done quicker. The problem here is that every time the GPU has to access this information, it has to do it through the AGP interface.

Then, first important information all resources should have is where it should be saved, in system memory or in GPU memory. As this decision must be made by the user it can be done using the following function call: the setUsage (ACD_USAGE usage) where ‘usage’ can be ACD_USAGE_STATIC if it is saved in the GPU memory or ACD_USAGE_DYNAMIC if it is saved in system memory.

Remember that all the Resources inherit from this class, so all of them have available the methods defined in the resource class.

Next important characteristic a resource should have is which kind of resource it is. This is important because there are some situations where we would work with the Resource type and it is need a way to discover which resource it is. This information cannot be modified as it is not possible to transform one resource to another, so the resource type is set by the ACD when it is
create and it cannot be done throw the interface. Otherwise ACD exports one method to obtain the resource type. This can be done using the `ACD_RESOURCE_TYPE getType()` method where the possible resource types are `ACD_RESOURCE_BUFFER`, `ACDRESOURCE_TEXTURE1D`, `ACD_RESOURCE_TEXTURE2D`, `ACDRESOURCE_TEXTURE3D`, `ACDRESOURCE_TEXTURECUBEMAP` and `ACDRESOURCE_RENDERTARGET`.

Finally there is one more characteristic a resource has. This characteristic is required due to how the Attila simulator manage the texture and renders target resources. When we were talking about the Attila simulator we have seen that there are two different ways how different kinds of data images are saved in the GPU. If they are textures they are saved using the Morton layout while if they are render targets they are saved using a different specific layout which is different; so they are not compatible. APIs such as DirectX allow using a texture mipmap as a render target and this is a problem for us, as internal formats are different. As the Attila simulator allows using texture defined using the render target layout (obvious using this format reduce the performance) it is need a way to now which layout a resource it is using. So last parameter allows us to specify which memory layout it is used. This can be set using the `void setMemoryLayout(ACD_MEMORY_LAYOUT layout)` function where the possible memory layouts are `ACD_LAYOUT_RAW`, `ACD_LAYOUT_TEXTURE`, `ACD_LAYOUT_RENDERBUFFER`.

**ACDBuffer**

Buffers are used to store vertex attributes such as the vertex position, the color, the texture coordinates, indices…

The ACDBuffer structure is quite simple as it doesn’t safe how this data is internally structured. Offset to the first element, stride between elements, the element type and so on is information that is not saved here.

Then the ACDBuffer only contains the raw data and all kind of operations to set and update this data.

The available functions are `void pushData(const acd_void* data, acd_uint size)` to append some amount ‘size’ of data ‘data’ to the current ACDBuffer. Another one, `void updateData(acd_uint offset, acd_uint size, acd_ubyte* data)` that updates some amount ‘size’ of data ‘data’ starting from an offset ‘offset’ from the begging of the ACDBuffer.

Other useful operations are `void resize(acd_uint size, bool discard)` that allows to resize the ACDBuffer size. ‘Size’ value is the new size of the ACDBuffer and the ‘discard’ value is used to choose between creating a buffer without any content of saving the previous content. The content saved is the first ‘size’ bytes from the begging of the ACDBuffer.

Finally there are some methods to clear the buffer content `void clear()` and get the data `acd_ubyte* getData()` or the size `acd_uint getSize()` of the ACDBuffer.
ACDTexture

The ACDTexture contains the texture raw data and some information about this texture data. There are many types of textures and Attila supports the follow ones: 2D, 3D (a.k.a. volume texture) and Cube Maps. In order to support all of them it has been defined a super class called ACDTexture that joins all the common characteristics all texture types have. Then for each specific texture type one new class is created which inherits from the main ACDTexture object. This structure is very logic as all texture types share some characteristics and it is also useful to the API programmer as he can reference whatever texture type using the ACDTexture type.

Textures share some characteristics but in fact they are quite different. The problem is each texture type has its own particular characteristics, for example the 3D texture type has the depth component while the Cube Map type the face attribute, and none of them are required in the 2D texture. So, know what are the common characteristics is not so easy.

Another thing to consider is that all texture types have mipmap levels. First of all, clarify that 2D and 3D mipmaps are exactly the same, the only difference is the amount of data needed to define them. The question is if these mipmap can be exposed in the interface. If they are exposed, the user can manage them directly, so they could do practically whatever they want with one mipmap object, including removing it, what would leave the texture in an inconsistent state. To avoid all this problems, the ACD doesn’t expose the mipmap levels to the user. Then, if this is done, other ways to manage texture mipmaps have to be provided, and this is done throw the ACDTexture object. Working this way allows the ACD to control all mipmap levels and strange situation cannot take place.

Other things that might be shared are texture characteristics such as the size and format among others. The question is if these characteristics are from the texture object or from each mipmap. It is obvious that each mipmap has its own size, so parameters like width, height and depth are inside the mipmap. The same happens to the format parameter. Current APIs only allow textures with all their mipmaps with the same format; although it could be possible that in the future each mipmap level has its own format.

Taking into consideration that mipmaps are not exposed to the user, that many of the characteristics are inside the mipmap object and that many of the functions calls have dependent parameters, practically nothing remains in the ACDTexture object. Inside the
ACDTexture object only remains functions to know how many mipmaps are set \( \text{acd_uint} \) getSettedMipmaps() and to set which is the minimum and maximum mipmap levels, \text{void} \) setBaseLevel(acd_uint minMipLevel) and \text{void} \) setMaxLevel(acd_uint maxMipLevel) that are going to be used.

From the basic ACDTexture object each texture type inherits. The available texture formats are ACDTexture2D, ACDTexture3D and ACDTextureCubeMap. Remember that Cube Map is a kind of texture composed by 6 2D textures built them up as a cube. Each one of this 6 2D textures is called face, and ACD define them as \text{ACD_CUBEMAP_POSITIVE_X} \text{ACD_CUBEMAP_NEGATIVE_X}, \text{ACD_CUBEMAP_POSITIVE_y}, \text{ACD_CUBEMAP_NEGATIVE_y}, \text{ACD_CUBEMAP_POSITIVE_Z} and \text{ACD_CUBEMAP_NEGATIVE_Z}.

![ACD_CUBEMAP_FACE](image)

Figure 28: This figure shows how each face is named by the ACD

All the ACDTexture types provide the same methods, but the difference between them is the parameters they have. To avoid explaining the same methods for each texture type they are only explained once, and when one of them have more parameters for certain types they are defined.

As the ACD interface doesn’t expose the Mipmap object, the only way to introduce them is using the ACDTexture interface. Then, the basic operations all texture types have are those to introduce and update the mipmap levels.

\[
\text{void \ setData( \ ACD_CUBEMAP_FACE \ face,} \\
\text{acd_uint \ mipLevel,} \\
\text{acd_uint \ width,} \\
\text{acd_uint \ height,} \\
\text{acd_uint \ depth,} \\
\text{ACD_FORMAT \ format,} \\
\text{const \ acd_ubyte* \ srcTexelData,} \\
\text{acd_uint \ texSize)}
\]
void updateData(ACD_CUBEMAP_FACE face,
                acd_uint mipLevel,
                acd_uint x,
                acd_uint y,
                acd_uint z,
                acd_uint width,
                acd_uint height,
                acd_uint depth,
                ACD_FORMAT format,
                const acd_ubyte* srcTexelData)

The setData method is used to add one mipmap level to a given texture. When a mipmap level
is added all its characteristics are provided in the function call. First important characteristic is
which mipmap level is going to be add, ‘mipLevel’, and in case the texture resource is a cube
map it is also required to know to which face the mipmap belongs to. Other information
necessary is the width ‘width’, height ‘height’ and the depth ‘depth’ (only with 3D textures), as
well as the texture format ‘format’ and the data itself ‘srcTexelData’. In case the texture format is
a compressed format it is also need the texture size.

The updateData method is used to update the data of a given mipmap. The parameters are
more or less the same, ‘face’ and ‘mipLevel’ to select the mipmap to update, and a region to
update. The region is specified with an initial point, ‘x’, ‘y’, ‘z’ (only with 3D textures) and the
width ‘width’, height ‘height’ and depth ‘depth’ (only with 3D textures) of the region. Other
information is a pointer to the data used to update the mipmap ‘srcTexelData’ and the format
‘format’ used with this data. This format must be the same as the current mipmap format.

Using the updateData method is not the unique way to modify the mipmap data. Once the
mipmap is created, there is another way to modify it and is using the following functions:

acd_bool map(ACD_CUBEMAP_FACE face,
             acd_uint mipLevel,
             ACD_MAP mapType,
             acd_ubyte*& pData,
             acd_uint& dataRowPitch)

acd_bool unmap(ACD_CUBEMAP_FACE face, acd_uint mipLevel)

The map function provides a pointer to the mipmap data space and allows the user to modify its
content depending on which is the ACD_MAP type selected. The ‘face’ and ‘mipLevel’
parameters are used to select the mipmap level of the texture resource. The ‘mapType’
parameter is used to select which kind of modifications the user is going to do on the mipmap
data. The available ACD_MAP types are:

ACD_MAP_READ: Only read operations
ACD_MAP_WRITE: Only write operations
ACD_MAP_READ_WRITE: It is possible to read and write
Finally ‘pData’ and ‘dataRowPitch’ are output parameters. ‘pData’ is the pointer to the mipmap data space and ‘dataRowPitch’ is the size in byte of one row of the mipmap.

This is useful as it allow the user to modify the mipmap data without having to update the data. Although it is a dangerous function as there is no control about how the user modifies the data, so it could damage the mipmap info or even worse, destroy other information that comes after the mipmap information.

The unmap function only needs the face and the mipmap level of the region to unmap.

Other available methods are getters to obtain information about a given mipmap. Remember that the face parameter is only need for cube map textures, and the getDepth function is only available for 3D textures.

```c
acd_uint getWidth(ACD_CUBEMAP_FACE face, acd_uint mipmap)
acd_uint getHeight(ACD_CUBEMAP_FACE face, acd_uint mipmap)
acd_uint getDepth(ACD_CUBEMAP_FACE face, acd_uint mipmap)
ACD_FORMAT getFormat(ACD_CUBEMAP_FACE face, acd_uint mipmap)
acd_uint getTexelSize(ACD_CUBEMAP_FACE face, acd_uint mipmap)
acd_uint getSamples(ACD_CUBEMAP_FACE, acd_uint mipmap)
```

Figure 29

**ACDRenderTarget**

The ACDRenderTarget resource is the simplest one. Once it has been created any modification can be done over it. The only functions calls available are getters to obtain information about how it has been created.
It is important to introduce that the ACDRenderTarget object internally contains a ACDTexture. The reason why it has been done this way is because the render target and the texture resource share many characteristics and also because some APIs create the render targets from texture resources, and then they allow to use this resource as a texture or as a render target.

Then, the getters available at the ACDRenderTarget in fact get the value from the ACDTexture resource. Then the ACDRenderTarget resource is nothing more than a different interface provided to a ACDTexture resource.

The available getters are the following ones:

acd_uint getWidth()
acd_uint getHeight()
ACD_FORMAT getFormat()
acd_bool isMultisampled()
acd_uint getSamples()
acd_bool allowsCompression()
ACDTexture* getTexture()

ACDShaderProgram

The ACDShaderProgram is a structure, provided by the ACD, used to save the shader program. Even having two different kinds of shader programs, vertex and fragment shaders, in this case it is not necessary to have one different structure for each one, as the only difference between each shader type is the instruction set it uses, and this is only important when the shader is executed by the shader unit. This decision is very important, because the new GPU architectures include new shader program types, and this structure can support them without any change.

First group of functions available in the ACDShaderProgram are those to set and get the shader program inside the ACDShaderProgram object. It is possible to provide the shader code using the Attila Byte Code or the Attila Assembler Code. Usually shader code is provide using the Attila Byte Code since each specific API uses one of the available shader translators at the ACDX to translate a shader program given in another shader language to the Attila Byte Code. Attila Assembler code is usually used by the specific API programmer when he needs to program any shader to emulate some capability. No matter which is the shader language provided by the user, the ACDShaderProgram generates the other, and both codes are available at the ACDShaderProgram object.

Then, to set the shader language inside the ACDShaderProgram there are available the void
setCode(const acd_ubyte* attilaByteCode, acd_uint sizeInBytes) to set a
shader code defined in Attila Byte Code or void setProgram(acd_ubyte *attilaASM) when the program is given using the Attila Assembler Code.

It is possible to get both codes using the const acd_ubyte* getCode() method to obtain the Attila Byte Code and const acd_ubyte* getProgram() to obtain the Attila Assembler Code. Remember that it doesn’t matter the shader language use to provide the shader, both are available. Finally with acd_uint getSize(), it is possible to obtain the size of the Attila Byte Code.

Apart from the shader code, the constant table accessed by the shader program is also saved inside the ACDShaderProgram object. Constants can be set using the void setConstant(acd_uint index, const acd_float* vect4) method, telling which constant it is being set ‘index’ and which is the new value ‘vect4’. This value must be 4 floats. Values from the constant table can be retrieved using void getConstant(acd_uint index, acd_float* vect4) method. The parameters are the same, but now the ‘vect4’ param return the value.

Apart from those methods, there are some operation to print the different shader program codes and the constant table which are useful to debug the code.

void printASM(std::ostream& os)
void printBinary(std::ostream& os)
void printConstants(std::ostream& os)

State management
Next important topic to describe is which structures the ACD interface provides to manage the state.

When the Attila simulator has been described, different units have been explained, and all of them must be set appropriately to run. Then, to every unit one structure is created to manage it and all this structures are controlled by the ACDDevice. The problem is that some of these structures are very small, so instead of having an independent object for them, they are added to the ACDDevice, reducing the number of objects.

Then the ACDDevice main task is manage some part of the state directly and manage the rest using the other structures.

To explain how the state management works, the ACDDevice structure is used because it controls all the structures and also contains a piece of the state. Also, to map what previous chapters have been explained, the graphic pipeline order would be followed.

**ACDDevice**

The device is the most important and complex piece of software of this project as it the responsible for organizing how all the state is synchronized with the GPU. As we previous said, some functionalities are managed by the device but others are divided in other classes. Otherwise, those functionalities that are in other classes are also controlled by the device.

The first important capability the ACDDevice offers is to create and destroy objects. Available resources have been defined before but we haven’t said anything about how they can be created and destroyed. The programmer cannot instantiate any new resource and every new resource it is desired must be created using the ACDDevice.

For every available resource the Device have one method to create it. To destroy whatever type of resource there is available one generic method; remember that all resources inherit from the Resource class. As the ACDDevice is the only element that can create and destroy objects it is possible to track all the objects that have been created or removed, improving how the resource management is done.

The available functions to create and destroy the different resources are the following ones. The simplest functions are those to create whatever texture type or an ACDShaderProgram. They don’t require any parameter and return the requested object.

ACDTexture2D* createTexture2D()
ACDTexture3D* createTexture3D()
ACDTextureCubeMap* createTextureCubeMap()
ACDShaderProgram* createShaderProgram()
When an ACDBuffer is created it is possible to introduce data inside it.

\[
\text{ACDBuffer* createBuffer(acd_uint size = 0, const acd_ubyte* data = 0)}
\]

This is done setting the appropriate pointer to the data in ‘data’, and the amount of bytes being introduced ‘size’. It is possible to create an empty ACDBuffer setting both values to 0. It is also possible to allocate a buffer of size ‘size’ without adding data; in this case ‘data’ must be set to 0.

As previously seen, ACDRenderTarget objects are created from ACDTexture objects. Then to create an ACDRenderTarget is used this function:

\[
\text{ACDRenderTarget* createRenderTarget( ACDTexture* resource, }
\text{ ACD_CUBEMAP_FACE face, acd_uint mipmap )}
\]

where the most important parameter is the ‘resource’ that is the texture resource used, and the ‘face’ and ‘mipmap’ level selected from this ACDTexture. The face parameter is only required if it is a cube map texture, otherwise it can be set to whatever value. The result of this operation is a ACDRenderTarget object that is bind with an ACDTexture. Remember that from now on the internal layout of the texture mipmap used in the render target as changed from the morton layout to the render target layout.

Finally to destroy whatever resource it is used the function \text{acd_bool destroy(ACDResource* resourcePtr)} with only the resource to destroy as a parameter.

Now, the next thing to explain is how the graphic pipeline is configured.

The first thing it is needed to configure in the ACDDevice is the streamers.

\textbf{ACDStream}

The Streamer is the unit responsible for reading the vertex data from memory. There are many ways how this data can be read and they depend on different parameters such as the type of data they hold, the offset to the first element, the stride between different elements and so on. So each ACDStream must be configured appropriately to be able to read the vertex data in the correct way.

The ACD provides an ACDStream structure for each stream unit the underlying hardware has.
ACDStream objects are created internally by the ACDDevice and when the programmer would like to access them he has to ask the ACDDevice to obtain the ACDStream object that is bind with the requested stream. This is done using the following function:

```cpp
ACDStream& stream(acd_uint streamID)
```

where the ‘streamID’ parameter can be a value between 0 and the maximum number of streams the simulator has been configured with.

Once the programmer has an ACDStream object is time to configure it. The most important information it has is an ACDBuffer object that holds vertex data information. Remember that the ACDBuffer only contains raw data, so to be able to read the data the ACDStream must know how data is structured, information that must be set in the ACDStream object.

So the information that is saved is the offset to the first element, the stride between elements, the type of data it is holding and how many components are packet together. Finally, there is another variable called frequency that is used for instancing. With this variable you can set which streams are used as index or as instances.

This schema shows how each parameter helps to read the buffer. As it can be seen, no more information is needed to read the ACDBuffer.

![Component Form](image)

Figure 31:

To set all this parameters the following functions are available in the ACDStream object.

The most important is `void set(ACDBuffer* buffer, const ACD_STREAM_DESC& desc)`. With this function it is possible to configure the entire ACDStream. The first parameter, ‘buffer’ is the ACDBuffer bind to the stream while the second parameter, ‘desc’ is an structure that contains all the parameters to read the buffer(offset, stride, type, the number of components and the instancing).

It is also possible to set all this parameters one by one using the following functions:

```cpp
void setBuffer(ACDBuffer* buffer)
void setOffset(acd_uint offset)
void setComponents(acd_uint components)
void setType(ACD_STREAM_DATA componentsType)
void setStride(acd_uint stride)
void setFrequency(acd_uint frequency)
```
It is also possible to retrieve the information from an ACDStream using a unique function to obtain it \code{void get( ACDBuffer* buffer, ACD_STREAM_DESC& desc)} or one by one using the following functions:

\begin{verbatim}
ACDBuffer* getBuffer()
acd_uint getOffset()
acd_uint getComponents()
ACD_STREAM_DATA getType()
acd_uint getStride()
acd_uint getFrequency()
\end{verbatim}

Another important thing to take into consideration is that the index buffers are not saved in whatever ACDStream. The ACD manage them in a different way and to set them it is necessary to use different methods. The index buffer is set in the ACDDevice using the \code{void setIndexBuffer( ACDBuffer* ib, acd_uint offset, ACD_STREAM_DATA indicesType)} method where it is required the ACDBuffer where the index info is saved ‘ib’, the offset to the first element ‘offset’ and type of the indices ‘indicesType’. No more information is need.

After reading the vertex attributes using the ACDStream the working unit of the system are the vertices. Next step is vertex shading. The most important information needed here is the ACDShaderProgram that contains the vertex shader code. Remember that this object includes all the information needed, the shader code in Attila Byte Code and the constant table the shader code access. The \textbf{ACD} provide this function \code{void setVertexShader(ACDShaderProgram* program)} to set the shader vertex program that it is going to be used. But before vertex shading stage could be done another thing must be done. It is need to match each shader input (vertex attribute) with the appropriate ACDStream. This mapping is done using the following ACDDevice calls to enable \code{void enableVertexAttribute(acd_uint vaIndex, acd_uint streamID)} and disable \code{void disableVertexAttribute(acd_uint vaIndex)} this mapping. The ‘vaIndex’ is the shader input ID while the ‘streamID’ is the ACDStream ID.

Then, for each vertex attribute used inside the shader code, it is need to map it with the appropriate ACDStream that holds the vertex information. In case this vertex attribute is not being used, it must be disabled. This is very important because if an ACDBuffer is not used but it is map to a vertex attribute the stream unit of the GPU will remain active performing no useful work and generating more traffic in the pipeline.
After vertex shading is time to select the primitive type it is going to be used. Remember that this information is needed by the Primitive Assembly stage to group individual vertices into primitive units. Setting the primitive type is done using the following ACDDevice function:

```cpp
void setPrimitive(ACD_PRIMITIVE primitive)
```

where the different primitive types the ACD supports are:

- ACD_TRIANGLES
- ACD_TRIANGLE_STRIP
- ACD_TRIANGLE_FAN
- ACD_QUADS
- ACD_QUAD_STRIP

ACDRasterizationStage

Although clipping and culling don’t belong to the raster stage some publications include them as previous steps before performing the rasterization, so it was decided to include them inside the ACDRasterizationStage class.

Culling is used to discard geometry that is facing or not the viewer depends how it has been configured. The rasterizer provides the following method to set the culling mode:

```cpp
void setCullMode(ACD_CULL_MODE cullMode)
```

Available cull modes are:

- ACD_CULL_NONE if cull mode is disabled
- ACD_CULL_FRONT in case it is desired to discard geometry which has a normal looking to the viewer
- ACD_CULL_BACK which do the same as ACD_CULL_FRONT but discarding those geometry which has a normal looking opposite to the viewer
- ACD_CULL_FRONT_AND_BACK which discards all the geometry. As previous seen in chapter 2, there are two ways to compute the vertex normal regarding how vertices are picked, this can be done clockwise or counter clock wise. This can also be set using the following function:

```cpp
void setFaceMode(ACD_FACE_MODE faceMode)
```

First thing before getting deeper in this topic is remind that not every API uses the same range to set the Z value. For instance, openGL uses as Z range [-1,1] while DirectX uses the [0,1]...
range. As clipping discards all the geometry that is outside the frustum area, it is required to set the Z range appropriately to avoid discarding geometry that is inside the frustum area. This must be taken into consideration and set appropriately in the simulator. The ACDRasterizationStage provides one function call to set which convention it is being used. This is done through the `void setD3D9DepthRangeMode(acd_bool mode)` method where if mode is true the simulator uses the D3D9 Z range definition, otherwise it is used the openGL one.

Next important parameter is the viewport size. It is important as the rasterizer maps from the NDC coordinates to the viewport ones, so it must know which size has the viewport to do the conversion. Viewport size is configured with `setViewport(acd_int x, acd_int y, acd_uint width, acd_uint height)` method. Another important point about the viewport is which corner of the viewport is the origin point (0, 0). OpenGL and DirectX differ in this point: in OpenGL it is in the lower left corner while in DirectX is in the upper left. This is important when geometry is transformed to viewport coordinates. Attila simulator supports both viewport definitions and it must be set using the `useD3D9PixelCoordConvention (acd_bool use)` function where use is true in case it uses D3D rules and false in case it follows openGL ones.

There are two different ways how the geometry can be rasterized, one only rasterizing the geometry borders and another rasterizing the entire surface the primitive define. Attila supports both ways of rasterizing and they can be set using the `setFillMode(ACD_FILL_MODE fillMode)` function where the available ACD_FILL_MODE are ACD_FILL_SOLID and ACD_FILL_WIREFRAME.

All graphic cards must be able to rasterize whatever API is on top of them without knowing it. The problem is that openGL and DirectX have some differences in their rasterization rules. These differences must be taken into consideration, so when the specific API on top of the ACD is using the same rasterization rules as D3D it must set the `useD3D9RasterizationRules(acd_bool use)` to true, otherwise it follows the openGL rules, use value is set to false.

Other parameters that can be configured inside the rasterizer is how vertex data is interpolated after the rasterization took place. Interpolation mode can be selected on each fragment data input using the `setInterpolationMode(acd_uint fshInputAttribute, ACD_INTERPOLATION_MODE mode)` method where fshInputAttribute is the fragment shader attribute to set and the available modes are ACD_INTERPOLATION_NONE, when this data is not interpolated, ACD_INTERPOLATION_LINEAR, ACD_INTERPOLATION_CENTROID...
and ACD_INTERPOLATION_NOPERSPECTIVE. The interpolation modes are not explained as they a little bit complex and are not need to understand the project.

After rasterization the text step is fragment shading, but before it, it is necessary to configure the samplers that are going to be used inside the fragment shader with the texture data that the shader is going to need.

Samplers (a.k.a Texture Units) are units that their main task is read texture data and apply to this data the required filters depending on how the texture is mapped on the geometry. So, the ACDSampler must contain all the information needed to read the texture and filter it.

**ACDSampler**

The ACDSampler behave more or less like the ACDStream does. The ACDDevice initializes as many ACDSampler objects as texture units the underlying hardware have. Then the API programmer can request to the ACDDevice any specific sampler and it return one ACDSampler object. This is done using the `ACDSampler& sampler(acd_uint samplerID)` method, where ‘samplerID’ is the required sampler unit ID.

Before talking about which characteristics a sampler unit has, it is better to clarify the difference between a sampler unit and a texture to understand why different information is associated with each other. Information associated with a texture, as have been seen, is those that are hardly related with how a texture is built up, while sampler unit contains all the information about how a texture is going to be used in this situation. One texture can be used in different samplers as each sampler has its own behavior.

Enable and disable a sampler is the first important functionality, and is done with the `void setEnabled(acd_bool enable)` function call. If a ACDSampler is disabled and a fragment shader tries to access it no result will be obtained, generally a black sample. Logically once the sampler is active it is need to set the texture it is going to be used. The `setTexture(ACDTexture* texture)` method set whatever texture resource.

Other parameters that can be set on an ACDSampler describe how the texture is going to be read and filtered. The texture address mode is used to configure what happens when the requested texel is outside the range [0,1]. This is also known as the wrapping mode. The available address modes are ACD_TEXTURE_ADDR_CLAMP, texture coordinate are clamp to 1, ACD_TEXTURE_ADDR_CLAMP_TO_EDGE, ACD_TEXTURE_ADDR_CLAMP_TO_BORDER, the border color is used when the texture coordinate is up to 1, ACD_TEXTURE_ADDR_REPEAT, in this case the texture coordinates when are bigger than 1 are restarted to 0, ACD_TEXTURE_ADDR_MIRRORED_REPEAT, this time, when the texture coordinate is up to 1
instead of restarting the value to 0, it starts decrementing to 0, and when the 0 is reached, it
starts incrementing another time, and so on. Address Modes can be set for each texture axis,
so multiple address modes can be used in one texture. They are set using the
setTextureAddressMode(ACD_TEXTURE_COORD coord, ACD_TEXTURE_ADDR_MODE
mode) where ‘coord’ is the axis it is being configured and ‘mode’ is the address mode set to
this axis.

Texture resources are not always ranged between [0,1] by all the APIs. Some of them use non
normalized coordinates, where the range is between 0 and the texture width, height or depth,
depending on the axis. Attila also supports this kind of referencing mode. It can be enabled
setting to true the void setNonNormalizedCoordinates(acd_bool enable) ‘enable’
value, otherwise normalized system is used.

Another important parameters are magnification and minification filters. As it has been seen,
they are used when the texture size and the region where it has to be map have a different size.
Available filters are ACD_TEXTURE_FILTER_NEAREST, ACD_TEXTURE_FILTER_LINEAR,
ACD_TEXTURE_FILTER_NEAREST_MIPMAP_NEAREST,
ACD_TEXTURE_FILTER_NEAREST_MIPMAP_LINEAR,
ACD_TEXTURE_FILTER_LINEAR_MIPMAP_NEAREST
ACD_TEXTURE_FILTER_LINEAR_MIPMAP_LINEAR, all of them explained in chapter 2. Filters
are set using the

void setMinFilter(ACD_TEXTURE_FILTER minFilter)

void setMagFilter(ACD_TEXTURE_FILTER magFilter)

methods, where ‘minFilter’ and ‘magFilter’ are the selected filters.

The API programmer cannot set which mipmap level is used because this is done by the
Texture Unit taking into consideration on which surface the texture have to be map on.
Depending on which mipmap level the Texture Unit has selected the sample obtained have a
different Level of Detail (LOD). The larger the mipmap level is the higher the LOD is. It is
possible to fit this LOD level between a range using the setMinLOD(acd_float minLOD)
and void setMaxLOD(acd_float maxLOD) methods to select a range between
[minLOD,maxLOD]. Then the Texture Unit not only will select the best mipmap level, but also
the mipmap level that provides a sample with a LOD between the range. It is obvious that the
LOD level is strongly related with the mipmap level.

setLodBias
setUnitLodBias
Another important topic to talk about is how a color is represented. Until now, it has been always use the RGB color space, where colors are represented using three components (red, green and blue) ranged between 0 and 1. But, there are more color spaces, and one of them is sRGB. The sRGB is not explained in this project but information about it can be found in (WIKIPEDIA). The ACD also support this color representation, but it must be active setting to true the setSRGBConversion(acd_bool enable) method.

Finally the texture unit supports another capability. The typical behavior of a texture unit is return a sample from a texture and some texture coordinates. But it is possible that, instead returning this sample, compares it with another value, and returns the result of the comparison. Then, instead of returning the texture sample, what it is returned is the result of the comparison between the texture sample value and a given value. The value being compared with the sample is provided using the $x$ component of the texture coordinate, so when this mode it is used, the format of the texture coordinates must have at least 4 components.

To enable this mode it is use the void setEnableComparison(acd_bool enable) method and to select the comparison function it is used the setComparisonFunction (ACD_TEXTURE_COMPARISON function) where the possible ‘function’ values are:

- ACD_TEXTURE_COMPARISON_NEVER: Always 0
- ACD_TEXTURE_COMPARISON_ALWAYS: Always 1
- ACD_TEXTURE_COMPARISON_LESS: 1 if sample < w
- ACD_TEXTURE_COMPARISON_LEQUAL: 1 if sample <= w
- ACD_TEXTURE_COMPARISON_EQUAL: 1 if sample == w
- ACD_TEXTURE_COMPARISON_GEQUAL: 1 if sample >= w
- ACD_TEXTURE_COMPARISON_GREATER: 1 if sample > w
- ACD_TEXTURE_COMPARISON_NOTEQUAL: 1 if sample != w
If OPT == '<'

```latex
\begin{tabular}{c|c|c|c}
R & G & B & OPT \\
0.3 & 0.1 & 0.9 & 0.2 & 0.2 & 0.2
\end{tabular}
```

Figure 35

ACDTexture

ACDSampler

Figure 36
Once the ACDSampler units have been configured next step is setting the appropriate fragment shader that is going to be used. As with the vertex shader program, fragment shader is set using an ACDSampler object. To set the fragment program the ACD provides the `void setFragmentShader(ACDSampler* program)` method.

After fragment shading different test to discard fragments must be done. Some of them are in different classes like the Z and the Stencil test, but others are configured directly on the ACDDevice.

Scissor test is configured using the ACDDevice. To configure the scissor test it is possible to enable and disable it using the `enableScissor(acd_bool enable)` function. Scissor test defines a rectangle area where all the fragments inside this area pass the test otherwise are rejected, so if scissor is enabled it is necessary to define this area. This is done setting the initial point, in viewport coordinates, and the width and height of the area. This is done using the function `setScissor(acd_int x, acd_int y, acd_uint width, acd_uint height)`

Apart from the scissor test, there are also the Z and Stencil test, but they are configured inside the ACDZStencilStage.

**ACDZStencilStage**

Z test can be enabled and disabled using the `void setZEnabled(acd_bool enable)` function. Comparison function can be set using `void setZFunc(ACD_COMPARE_FUNCTION zFunc)` where available `ACD_COMPARE_FUNCTION` are:

- `ACD_COMPARE_FUNCTION_NEVER`: Return always false
- `ACD_COMPARE_FUNCTION_LESS`: Source < Destination
- `ACD_COMPARE_FUNCTION_LESS_EQUAL`: Source <= Destination
- `ACD_COMPARE_FUNCTION_EQUAL`: Source == Destination
- `ACD_COMPARE_FUNCTION_NOT_EQUAL`: Source != Destination
- `ACD_COMPARE_FUNCTION_GREATER`: Source > Destination
- `ACD_COMPARE_FUNCTION_GREATER_EQUAL`: Source >= Destination
- `ACD_COMPARE_FUNCTION_ALWAYS`: Return always true

It is possible to block writing the new Z value to the Z buffer when the Z test is passed, then the old Z value remains in the Z buffer for further tests. This is done activating the ZMask. When it is
active no Z buffer updates are made even the fragment passes. To active this it is used the

```c
void setZMask(acd_bool mask)
```

function.

Stencil test is a little bit more difficult than the Z test. How it works have been explained in the chapter 2. First of all, a enable/disable function is available to active the test; 

```c
void setStencilEnabled(acd_bool enable)
```

The reference value, comparison function and the ADD mask are set using the

```c
void setStencilFunc(ACD_COMPARE_FUNCTION func, acd_uint stencilRef, acd_uint stencilMask )
```

where the ACD_COMPARE_FUNCTION values have been explained before.

To set the behavior of the result of the stencil test the

```c
void setStencilOp(ACD_STENCIL_OP onStencilFail, ACD_STENCIL_OP onStencilPassZFail, ACD_STENCIL_OP onStencilPassZPass)
```

method is used. ‘OnStencilFail’ is the operation set when the Stencil test fails, ‘onStencilPassZFail’ is the operation set when the stencil test is passed and the Z test fails and ‘onStencilPassZPass’ is when both, Stencil and Z tests, are passed. Available operations are the following ones:

- ACD_STENCIL_OP_KEEP: Keep the existing stencil data
- ACD_STENCIL_OP_ZERO: Update the stencil buffer to 0
- ACD_STENCIL_OP_REPLACE: Set the stencil Data to the reference value
- ACD_STENCIL_OP_INCR_SAT: Increment the stencil value by 1 and clamp the result
- ACD_STENCIL_OP_DECR_SAT: Decrement the stencil value by 1 and clamp the result
- ACD_STENCIL_OP_INVERT: Invert the stencil buffer data value
- ACD_STENCIL_OP_INCR: Increment the stencil value by 1, and wrap the result if necessary
- ACD_STENCIL_OP_DECR: Increment the stencil value by 1, and wrap the result if necessary

Before writing the final value to the stencil buffer it is possible to apply an ADD mask to this value. This is done using the

```c
void setStencilUpdateMask (acd_uint mask).
```

Fragments at this point are those that have passed all the test and are going to be written to the frame buffer. There are many ways how this can be done, and that is what blending stage controls. But remember that the frame buffer is nothing more than one render target that it can be changed, something that are going to be explained later. Attila also supports multi render target (MRT), what this means is that fragment can be written in more than one render target at the same time. The maximum number of render targets depends on how the simulator is
configured, but each one is identified using and ID from 0, the main render target, to N which is the maximum number of render targets together. Then let's see how blending is controlled.

**ACDBlendingStage**

Blending is the stage where it is controlled how input fragments colors are combined with the color in the render target.

Before the ACDBlending stage can be done it is need to set the ACDRenderTargets. This is done using the `acd_bool setRenderTarget(acd_uint indexRenderTarget, ACDRenderTarget *renderTarget)` where 'indexRenderTarget' is the render target ID and the 'renderTarget' object that is going to be set. In case it is desired to disable one of the MRT, it is necessary to use the same function but with the 'renderTarget' value to 0.

From now on, all the operations the ACDBlendingStage has have a parameter that is the MRT it is being configured.

To work with the ACDBlendingStage, the first thing is necessary to do is enable the blending stage for each MRT. This is done using the `void setEnable(acd_uint renderTargetID, acd_bool enable)` function where 'renderTargetID' is the render target unit being configured.

Blending explicitly splits how the RGB component and the Alpha component are configured. So each blending functions always has its version to the RGB component and another for the Alpha component. Also remember that source color is referred to the color that the input fragment has and the destiny color is the color of the fragment that is already in the render target.

So blending starts from two different colors, the source color and the destiny color, then let's see how blending allow us to combine them.

First thing it can be done is multiply each color by a factor. This factor is not introduced as a number; it is compute using a blending option which is the value introduce by the user. Remember that there is one factor for the RGB color and another for the Alpha component. To set this factor the following functions can be used:

```c
void setSrcBlend(acd_uint renderTargetID, ACD_BLEND_OPTION srcBlend)
void setDestBlend(acd_uint renderTargetID, ACD_BLEND_OPTION destBlend)
void setSrcBlendAlpha(acd_uint renderTargetID, ACD_BLEND_OPTION srcBlendAlpha)
void setDestBlendAlpha(acd_uint renderTargetID, ACD_BLEND_OPTION destBlendAlpha)
```

where the first parameter is which render target unit is configured and the second refers to the way how the factor is going to be calculated.
Some ACD_BLEND_OPTION use a default blend color to compute the factor. By the time let’s introduce that blend color is set using the `void setBlendColor(acd_uint renderTargetID, acd_float red, acd_float green, acd_float blue, acd_float alpha)` function to introduce one blend color to each render target.

Then, the way the factor is compute is using one of the available ACD_BLEND_OPTIONS:

- **ACD_BLEND_ZERO**: Factor components are 0
- **ACD_BLEND_ONE**: Factor components are 1
- **ACD_BLEND_SRC_COLOR**: Factor components are obtain from the source color
- **ACD_BLEND_INV_SRC_COLOR**: Factor components are 1 minus the source color components
- **ACD_BLEND_SRC_ALPHA**: Factor components are always the source alpha component
- **ACD_BLEND_INV_SRC_ALPHA**: Factor components are always 1 minus the source alpha component
- **ACD_BLEND_DEST_COLOR**: Factor components are obtain from the destiny color
- **ACD_BLEND_INV_DEST_COLOR**: Factor components are 1 minus the destiny color components
- **ACD_BLEND_DEST_ALPHA**: Factor components are always the destiny alpha component
- **ACD_BLEND_INV_DEST_ALPHA**: Factor components are always 1 minus the destiny alpha component
- **ACD_BLEND_SRC_ALPHA_SAT**: 
- **ACD_BLEND_CONSTANT_COLOR**: Factor components are the default blend color ones
- **ACD_BLEND_INV_CONSTANT_COLOR**: Factor components are 1 minus the default blend color ones
- **ACD_BLEND_CONSTANT_ALPHA**: Factor components are the default blend color alpha one
- **ACD_BLEND_INV_CONSTANT_ALPHA**

After each factor is applied to the source and the destiny color, next step is combining them. This is done setting the following blending functions:

```c
void setBlendFunc(acd_uint renderTargetID, ACD_BLEND_FUNCTION blendOp)
void setBlendFuncAlpha(acd_uint renderTargetID, ACD_BLEND_FUNCTION blendOpAlpha)
```

where, the first function is to set the blend function for the RGB component and the second one for the alpha component.

The first parameter is the render target unit configured and the second one the blending function used. The available ACD_BLEND_FUNCTIONs are:

- **ACD_BLEND_ADD**: Source plus destiny
- **ACD_BLEND_SUBTRACT**: Source minus destiny
- **ACD_BLEND_REVERSE_SUBTRACT**: source reversed minus destiny
ACD_BLEND_MIN: minimum between source and destiny
ACD_BLEND_MAX: maximum between source and destiny

From the above operation the final color is obtained. Otherwise, it is possible not to write all of them in the render target as it is possible to block specific color components. This can be done using the void setColorMask(acd_uint renderTargetID, acd_bool red, acd_bool green, acd_bool blue, acd_bool alpha) function, where red, green, blue and alpha are set to true if they can be written to the render target or false if they cannot.

It is possible to disable all of them using another function called void disabledColorWrite(acd_uint renderTargetID), which disables all the color components, so nothing is written in the render target.

Up here, these are all the steps necessary to configure the graphic pipeline in the ACD to draw a batch. Next sections explain other operations provided by the ACDDevice.

Clear operations

Another important set of operations are those used to clear a surface and set a value to it. These operations can be done on the Z buffer, the Stencil buffer or on an ACDRenderTarget.

All the clear operations are provided by the ACDDevice, and the parameter all of them need is the value used to clear the surface.

To clear the Z buffer or the stencil buffer it is used the void clearZStencilBuffer(acd_bool clearZ, acd_bool clearStencil, acd_float zValue, acd_int stencilValue) method. This function is used for both buffers. If they are being clear or not is set using the ‘clearZ’ for the Z buffer and the ‘clearStencil’ for the stencil buffer. Then, ‘zValue’ is the value used to clear the Z buffer and ‘stencilValue’ for the stencil buffer.

It is also possible to clear the render target. The ACDDevice only allows clearing the ACDRenderTarget set at the render target slot 0. To clear it, the void clearColorBuffer(acd_ubyte red, acd_ubyte green, acd_ubyte blue, acd_ubyte alpha) function is used, where the ‘red’, ‘green’, ‘blue’ and ‘alpha’ is the color used to clear the render target.

Clear operations don’t necessary need to clear the entire surface, it is possible to clear only a region of these surface. This is done enabling the scissor test and configuring it with the area that is desired to clear. Then, when the clear operation is issue, it will only clean the area set.
**Draw calls**

At his point all the possible ways how the ACD allows configuring the graphic pipeline have been explained. After setting the required parameters it is time to draw the scene using them. The ACD provide two different draw calls. One of them is used when no index buffer is set besides there is another draw call used when indices are used.

First of all, `void draw(acd_uint start, acd_uint count, acd_uint instances)` is used when no index buffer is set. Parameters are, from left to right, from which vertice it starts to draw ‘start’, the number of vertices to draw ‘count’ and how many instances it uses ‘instances’.

The indexed draw call is the following:

```cpp
class AcDdevice {
public:
    virtual void drawIndexed(acd_uint startIndex,
                              acd_uint indexCount,
                              acd_uint minIndex,
                              acd_uint maxIndex,
                              acd_int baseVertexIndex,
                              acd_uint instances) = 0;
};
```

Before using it, it is required to set the appropriate index buffer as it was told to do when ACDStreamer was explained. Notice that although the index buffer is set, if the draw call method is not the indexed one, it won’t be used. Parameters of this method, from left to right, are the first index buffer position which is going to be read ‘startIndex’, the number of indices read ‘indexCount’, from the range defined by the previous two values only indices between ‘minIndex’ and ‘maxIndex’ are accepted. ‘BaseVertexIndex’ is some value added to every index and ‘instances’ is the same as the previous call.

Here you can find an example of how the indexed draw call works:

Issuing this index draw call

```cpp
ACDDevice->drawIndexed(6, // startIndex
                      9, // indexCount,
                      3, // minIndex
                      7, // maxIndex
                      20 // baseVertexIndex
                      0, // instances
); 
```

being the primitive type the ACD_TRIANGLELIST and with this index buffer: {1,2,3, 3,2,4, 3,4,5, 4,5,6, 6,5,7, 5,7,8} it draws 3 triangles (indexCount = 9 and primitiveType= ACD_TRIANGLELIST), starting with index number 6 (startIndex).
So it ignores the first two triangles and the last one, only using the following indices: \{3,4,5, 4,5,6, 6,5,7 \}. Then the ‘baseVertexIndex’ is added to every index, being the new indices \{23,24,25, 24,25,26, 26,25,27\}. Then, this are the indices used to fetch the appropriate vertices.

**Copy operations**

Another capability the ACD also provides is copy operations to copy image resources between them. The methods provided by the ACD allow the user to copy the mipmap data to another mipmap, to copy a render target to a mipmap and vice versa. This section explains each copy operation can be executed.

The first operation is

```c
void copyMipmapToMipmap ( ACDTexture* inTexture,
                          acdlib::ACD_CUBEMAP_FACE inFace,
                          acd_uint inMipmap,
                          acd_uint inX,
                          acd_uint inY,
                          acd_uint inWidth,
                          acd_uint inHeight,
                          ACDTexture* outTexture,
                          acdlib::ACD_CUBEMAP_FACE outFace,
                          acd_uint outMipmap,
                          acd_uint outX,
                          acd_uint outY,
                          acd_uint outWidth,
                          acd_uint outHeight,
                          ACD_TEXTURE_FILTER minFilter,
                          ACD_TEXTURE_FILTER magFilter)
```

That allows the user to copy the mipmap data from one mipmap to another. This method is very flexible and allows the user to copy a region of a given size from a source mipmap to another different region that can have whatever other size of a destiny mipmap. The parameters are from up to down, the input texture resource ‘inTexture’, the ‘face’ (only in case it is a ACDTextureCube Map), the ‘mipmap’ level, ‘inX’ and ‘inY’ are the starting point from the source region to copy and ‘inWidth’ and ‘inHeight’ define how big is this region. Of course the region to copy cannot be outside the mipmap being copied. Next parameters are the same but referring to the destiny mipmap. Remember that the destiny region can have a different size from the input one. Finally it is possible to use different filters to avoid that the resizing from one region to the other harm the image, to this end ‘minFilter’ and ‘magFilter’ are used.

Next method is used to copy from a texture resource (by the time only a 2D texture) to a render target.

```c
void copyMipmapToRenderTarget ( ACDTexture2D* inTexture,
                                acdlib::ACD_CUBEMAP_FACE inFace,
                                acd_uint inMipmap,
                                acd_uint inX,
                                ```
```
acd_uint inY,
acd_uint inWidth,
acd_uint inHeight,
ACDRenderTarget * outRenderTarget,
acd_uint outX,
acd_uint outY,
acd_uint outWidth,
acd_uint outHeight,
ACD_TEXTURE_FILTER minFilter,
ACD_TEXTURE_FILTER magFilter)

The parameters required in this case are more or less the same. The main difference is that the source is an ACDRenderTarget, so the face and the mipmap parameters are not required.

Finally there is another method to copy from a render target to a given mipmap.

```
void copyRenderTargetToMipmap (
    ACDRenderTarget * inRenderTarget,
    acd_uint inX,
    acd_uint inY,
    acd_uint inWidth,
    acd_uint inHeight,
    ACDTexture * outTexture,
    acdlib::ACD_CUBEMAP_FACE outFace,
    acd_uint outMipmap,
    acd_uint outX,
    acd_uint outY)
```

The main difference here is that the destiny region must have the same size as the source region. Then the input is the source render target and the region to copy and the destiny is the texture, the face and the mipmap level and when the destiny region begins 'outX' and 'outY'.

**Debugging tools**

Another important topic to talk about is which tools the ACD interface offers to debug itself and to help the specific programmer to debug his specific APIs.

The ACD offers two different functions. First of all there is a function which dumps all the ACD state that it has been described in this chapter.

This can be done using the following function:

```
void DBG_dump(const acd_char* file, acd_enum flags)
```

where the ‘file’ parameter is the name of the file where to output the data and ‘flags’ is used to configure how the information is output. By the time, the ‘flags’ parameters is not available.

Here you can find an example of an output:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```
Another useful function to debug is `void DBG_printMemoryUsage()` which outputs how much memory is being used. It is useful because it can be used to configure appropriately the simulator.

The output of this function is:
The previous section has explained the ACD interface and which are the capabilities it offers to the API programmer. With the information provided whatever person can build a specific API on top of it without knowing anything else from the ACD.

This chapter is focused in explaining how is designed the internal structure to build the ACD interface. The explanation is divided in two different chapters; this first chapter explains the internal structure of the ACD and the next chapter is focused on explaining some interesting implementation issues.

All this information is useful for the people who would like to add or modify things inside the ACD.

Before continuing it is important to remember some of the basic requirements the ACD must have. The ACD is the responsible for allocating and deallocating the different resources. It also must try to reduce the AGP bus overhead to avoid being the bottleneck of the system. Remember that the bus traffic is generated when information is transferred from the main system memory to the GPU memory, and this usually happens when GPU registers are written or when the resources are synchronized. Finally the ACD should also take care that it is correctly synchronized with the GPU.

To be able to support all these requirements, new structures have been added to the ACD. To explain them, this chapter has been divided in three different sections. First sections describe how the ACD is built, highlighting the main topics in each area but not explaining them with too
much detail. Next to sections, explain more in detail the ACD structure focusing in the resource structure and the state structure.

**Internal design**

The internal design of the ACD resembles a lot the interface design. This is logic as the interface is nothing more than a set of classes that export some operations to the user, and then these operations must be implemented. So, it is logic to follow the same design.

![Class Diagram](image)

**Figure 37: Class diagram of the ...**

The following class diagram show how both structure are connected. For each class of the interface a new one is created and inherit from the interface. This new class contains the implementation of the methods discribed in the interface, as well as, all the resources and internal operations needed to provide these methods. The relations between the classes that belong to the internal design are the same as the relations between the classes in the interface.

Apart from the old structures a new one have appeared. This is the MemoryObjectAllocator. The main function of this class is synchronize appropriatelly the resources to the GPU, taking care that this must be done improving the GPU bus. This class is connected with all the classes that contain resource elements.
From the point of view of the resources, more or less have happened. The internal design resembles a lot the interface design, but this time more information have been added.

This class diagram show how from every type of resource defined in the interface a new one is created and inherits from the interface. Apart from inheriting from the interface, each resource inherits from another object called MemoryObject. This new object is very important as it includes all the information about the state of the resource between the ACD and the GPU. These classes implement the methods described in the interface and contain the structures and the internal methods necessary to implement them.

To implement the ACDTexture resource more classes are needed. Remember that each texture type is built using mipmap levels that are not exposed in the interface. Then, to accommodate the mipmap levels, internally each texture type is bind with a TextureMipmapChain, that is nothing more than a list with some specific operations to manage mipmaps, and this TextureMipmapChain contains a list of all the mipmap objects.
Before explaining each topic in depth, it is necessary to reanalyze which functions the ACDDevice manage. Remember that not all the state is managed by each specific structure, and some of them are managed by the ACDDevice.

The ACD interface doesn’t allow the specific API programmer to configure all the GPU stages. For example the Hierarchical Z or the early Z tests cannot be configured by the user. They are available in the underlying hardware, but is the ACD who configures this two units. The reason why this is done this way is because this test can only be enabled in certain situations to avoid graphic errors. It is possible to let the responsible of configuring them to the upper levels, but as the rules to enable and disable both tests are always the same; in order to simplify the work it is better to place them inside the internal design of the ACD.

This figure shows how the graphic pipeline is implemented in the ACD internal design. The red boxes are those implemented inside the ACDDevice, while the green ones are those
implemented in a different object. New elements added are the Early Z and Hierarchical Z stages. Outside the ACDDevice there is the MemoryObjectAllocator. Finally all the available resources are on the left side, parallel to the stage where they are used.

Resource structure

This section focuses on explaining more in depth which is the internal structure of each resource.

MemoryObject

When the ACD interface was explained, all the available resources inherit from the Resource class, which contains all the characteristics that are common among all the resources. Through all the information the Resource has, it doesn't have anything that binds the resource data with the GPU memory. This fact is logic as the user of the ACD doesn't need to know if the resource is synchronized with the GPU as this is the task of the ACD. Then, it is necessary to add this information to be able to synchronize the resources.

There are 4 different resources available: the ACDBuffer, the ACDTexture, the ACDRenderTarget and the ACDShaderProgram, but all of them have something in common, they contain some amount of data that must be synchronized with the GPU and many other information that describe how the data can be read. It doesn't matter which kind of data it is, all the data is placed the same way in the GPU. The basic work unit to work with the GPU memory is the memory region. A memory region can be defined as a memory area with a certain size that is contiguous in memory.

The first question here is how it is possible to match this data with the memory region. The most important problem that appears working with memory is the fragmentation. This problem takes place when small memory regions cannot be used because it is not possible to fit inside them new data. The easiest solution to minimize this problem is trying to use memory regions as small as possible. How smallest can be a memory region is determined by the need of the data to be contiguous in memory.

The problem is that many of the resources cannot be divided. The ACDBuffer and the ACDShaderProgram are resources that contain a program or a buffer of data. These two data types must be contiguous in memory, so the smallest region that can be used is one with the size of the resource to allocate. Otherwise, ACDTexture is made up of some mipmap levels. The mipmap levels from one texture don't have any relation between them, so they can be saved in different regions. This decision is logic as the GPU selects a given mipmap depending
on the surface where it has to be map, and then they have no relation. Each mipmap level cannot be divided in different regions as they have to be saved contiguous in memory. Finally, as the ACDRenderTarget resource is built up from one mipmap level of a texture, it also must be saved in one memory region.

The way memory regions are managed is the same for all the resources, so it is logic to create a new object call MemoryObject with all the information about the regions. Then, this object is inherited from every resource.

The following schema is an example of how an ACDTexture resource is mapped to the GPU memory.

![Diagram of memory mapping](image)

**Figure 40**

The MemoryObject is a class that contains information about all the memory regions bind to the resource. This is important, there is not a MemoryObject for each region, there is a MemoryObject for each resource, and the information of all the memory regions is contained inside it.

One important characteristic of the memory regions is that in some situations the memory regions in the resources might not be synchronized in the GPU. This happens because when the data of the resource is updated, this update is not synchronized immediately with the GPU, as this is done when a draw call is issue. This is done this way to minimize the traffic of the AGP bus.

To manage the state of each memory region, the MemoryObject has a structure that tracks the current status of a region. This structure is defined as follow:

```c
struct MemoryObjectRegion
```

---

92
First of all the ‘state’ shows the state of the region between the API memory and the GPU memory. The available statuses are:

MOS_Sync, is used when the information between the API and GPU is already synchronized.
MOS_ReAlloc, the memory region must be fully reallocated in another region. Usually this happens when the resource size is increased, and the new size is bigger than the previous memory region allocated.
MOS_NotSync, the memory region is not synchronized with the GPU. The data can be synchronized in the current memory region.
MOS_Blit, is used to indicate that the memory region has been updated in the GPU due to a blit operation, so API data is not synchronized with the GPU one.
MOS_RenderBuffer, this memory region is used to store a render target, so it is modified by the GPU and these changes are not synchronized with the API.

Many times, when a resource is updated, not the entire data is modified; usually only a small part. Then it is not necessary to update the whole region. The ‘firstByteToUpdate’ and ‘lastByteToUpdate’ are used to specify the memory range updated. Then when the resource must be synchronized, only the update region is synchronized. The ‘reallocs’ attribute is used to know how many times the region has been reallocated.

The MemoryObject class offers a great variety of methods to modify the state of a region.

void postBlit(acd_uint region);
void postRenderBuffer(acd_uint region);
void postUpdate(acd_uint region, acd_uint startByte, acd_uint lastByte);
void postUpdateAll();
void postReallocate(acd_uint region);
void postReallocateAll();

But what it is important to understand is when it is necessary to change the state of a region. The primary state of a region is MOS_Sync. This state ensures that data in the GPU is consistent. But when this data is updated in the API side, the state must be changed. There are two possible options, if the update fits in the current region size, the state is changed to MOS_NotSync, otherwise, if the update needs a bigger region, it is changed to MOS_ReAlloc. The MOS_Blit and MOS_RenderBuffer states are special, as they are only used by texture resources and also because when a texture resource is changed into one of these states it is not possible to change them. This is this way because this state implies that data stored in this
region has been modified by the GPU, so it is not synchronized with the API version, but this
time the newer information is saved in the GPU. Due to the GPU doesn’t provide any operation
to read this data from the GPU, it is not possible to modify those resources from the API.

Apart from those methods, there is one more than must be explained.

```c
acd_ubyte* memoryData(acd_uint region, acd_uint& memorySizeInBytes)
```

This is an abstract function that must be implemented by all who inherits the MemoryObject
class. It is very important because it is going to be used by the MemoryObjectAllocator which is
the element that tracks if all the elements are synchronized. This function is used to obtain the
data saved in a memory region. In case it is an ACDBuffer or an ACDShaderProgram, there is
only one region available, so all the data is provided. Otherwise, ACDTexture objects have one
region for each mipmap, so for every mipmap this function must be called, being the ‘region’ the
mipmap level desired and the ‘memorySizeInBytes’ a parameter that returns the size of the
memory region. In the case of an ACDBuffer or an ACDShaderProgram the region is always 0.

Finally, it is important to talk about how the regions are created. The MemoryObject provide two
different protected functions, which can only be used by each specific resource that inherits
from the MemoryObject class to create and delete a region. These functions are:

```c
void defineRegion(acd_uint region);
void undefinedRegion(acd_uint region);
```

and both of them require the number of the region ‘region’ that is created. Then, for example,
when a ACDTexture2D is created, it doesn’t have any region, but when a new mipmap is
added, internally the ACDTexture2D, apart from creating a new Mipmap object, it is defined a
new region to this mipmap level.

**ACDBufferImp**

The internal structure of the ACDBuffer resource is the simplest one. It only saves raw data.
Apart from that, nothing more is saved. Remember that the information about how to read the
buffer content is saved inside the ACDStream structure, then, any other attribute is needed
inside the ACDBuffer object.

As previously seen, the data of the ACDBuffer is contiguous in memory, so it must be saved
inside a unique memory region.

The problem with the ACDBuffer is that many times its size changes. Changing the size of the
resource usually implies having to reallocate the full memory region, and this is a problem as
increases the bus traffic. To solve this problem a new strategy is followed. Instead of allocating
a memory region of exactly the same size the current buffer, what is done is create a larger
memory region. Now, small updates don’t require an update, only when the buffer limit is
reached, the buffer must be reallocated. The problem with this solution is how much more
space it is needed. Allocating too much extra space causes that many memory areas cannot be
used. To avoid this problem what is done is use a growth factor. This factor is multiplied by the size of the data inside the buffer and it is used to determine how much bigger the buffer must be than the real data. Every time the buffer is reallocated this value is increased. So, if the memory region is reallocated many times, this factor grows very fast and increase the available space, otherwise the extra space is very small. The value is initialized to 1.

**ACDTextureImp**

In the previous chapter it has been explained that the interface exports an ACDTexture class with a few simple functions and from it each specific texture types inherit. The ACD internal design implements each specific texture type. As all the texture types have more or less the same function calls, the difference between them is the parameters, instead of explaining all the functions for each texture type, this is done only once and it is highlight which are parameters are specific for a texture type.

First of all, remember that when the ACD interface has been explained, many of the texture characteristics are saved inside each mipmap level. In fact each ACD texture object only have to save how many mipmaps a Texture have, which is the first mipmap level and the maximum mipmap level and a structure where to save the mipmaps.

Mipmap levels are saved in a structure called TextureMipmapChain which contains all the available mipmaps of the texture. This structure provides all necessary operations to add, remove, and obtain a mipmap level.

Inside the TextureMipmapChain what is saved are the Mipmap objects. Each object contains the data of the mipmap, and all the information about it, basically the width, the height, the depth (only for 3D textures), the format among and the mipmap size among others previously explained.

Finally it is important to remember that each mipmap level has its region, so for every mipmap added it is necessary to define a new region.
Texture Adapter

Working inside the ACD with the different texture types is a problem, as many times it is need to know which texture type you are working with to issue the appropriate call. Remember that many of the methods cannot be included inside the ACDTexture object because the same function call in different texture types has different parameters.

This is annoying from the point of view of the programmer, so to solve the problem a new class is introduced, the TextureAdapter.

The TextureAdapter only hides the different function definitions each texture type has. So the programmer has for each function type one unique function. Then the texture adapter discovers which kind of texture it is being used and issues the appropriate call.

This is an example of how the Texture adapter works:

```cpp
ACDTexture2D tex2D;
ACDTexture3D tex3D;
ACDTextureCubeMap texCM;
textureAdapter adapterTex2D (tex2D);
textureAdapter adapterTex3D (tex3D);
textureAdapter adapterTexCM (texCM);

adapterTex2D->getWidth(ACD_CUBEMAP_FACE face, acd_uint.mipmap);
adapterTex3D->getWidth(ACD_CUBEMAP_FACE face, acd_uint.mipmap);
adapterTexCM->getWidth(ACD_CUBEMAP_FACE face, acd_uint.mipmap);
```
The ACDRenderTarget is nothing more than a container of an ACDTexture object. In fact, it only contains a pointer to an ACDTexture object and the face and the mipmap level of the mipmap used as a render target. The ACDRenderTarget interface has many other operations to know the size, the format or other things of the render target. In fact, all this queries are bypassed to the ACDTexture resource, so, none of this information is inside the render target resource.

**ACDShaderProgramImp**

The ACDShaderProgram saves the shader program. Remember that the shader program is saved in two different versions, the Attila Assembler Code and the Attila Byte Code. Apart from the shader code, the constant table is also saved here, and also the size of the Attila Byte Code.

**State structure**

The internal state structure of the ACD is very similar to the interface structure. The state of the ACD is nothing more than a set of variables with some values. These values are set using the appropriate function calls the interface offers. To save all this variables some sort of data structures are used. There are many of these variables, and explaining all of them is not useful as they are very simple.

This section skips how the ACD saves internally the GPU state and focuses on how the state is synchronized with the GPU and also it explains some optimizations to reduce the traffic of the AGP bus.

**Synchronization**

How the different resources and the ACD state are synchronized with the GPU is one of the most, if not the most, important task the ACD has to manage with. It is important because it is a low level task, so it must be done with accuracy and avoiding extra work.

This section is divided in two different parts. The first one talks about how the resources are allocated using the MemoryObjectAllocator and the second one talk about how the state is synchronized with the GPU. The reason for this division is because the way this is done in both cases is different.

**Resource Allocation**
As previously seen, textures, buffers and shaders are in fact raw data that must be allocated inside the GPU memory space, which can be the GPU memory or the system memory accessible by the GPU.

The problem with these resources is that they can be updated, totally or partially. This is a problem because one of the basic requirements of the ACD is to minimize the amount of data transferred to the GPU. Then the ACD must take care of all these situations.

Before continuing explaining how the ACD manage the resources, it is need to know which functions the Driver offers to manage the memory.

The most important methods the Driver offer are those to allocate and deallocate GPU memory. Those methods are:

```c
u32bit obtainMemory(u32bit sizeBytes, MemoryRequestPolicy memRequestPolicy)
void releaseMemory(u32bit md);
```

The obtainMemory function is used to allocated one memory region with some size. The parameters are the number of bytes requested ‘sizeBytes’ and the kind of memory requested ‘MemoryRequestPolicy’. The available memory types are GPU memory ‘GPUMemoryFirst’ or system memory ‘SystemMemoryFirst’. It is important to highlight that the memory kind returned would be the requested type only in case it is available, if not the driver tries to obtain memory from the other type. One important thing is that the Driver doesn’t provide pointers to the memory regions allocated. Instead of this, it offers memory descriptor (MD) that reference the allocated space.

The releaseMemory function is used to deallocate a memory region from the GPU. This is done providing the MD of the region you want to deallocate.

Once you have allocated a memory region you can write on it. The only information you need is the MD descriptor of the region to write and use one of these functions:

```c
bool writeMemory(u32bit md, const u8bit* data, u32bit dataSize, bool isLocked = false);
bool writeMemory(u32bit md, u32bit offset, const u8bit* data, u32bit dataSize, bool isLocked = false);
```

The first one writes into the memory region pointed by the MD starting from the begging while the second method starts writing the memory area from the begging plus a given offset.

These two methods allow writing whatever thing in this memory region, texture mipmaps, vertex buffers and shaders.

About the shader, it is important to remember one special characteristic. When the Attila architecture has been explained, it was said that shader programs cannot be executed from
whatever GPU memory region; some memory space is reserved to hold the shaders that are going to be executed.

Shaders, as whatever other resource, must be saved in whatever memory area, and only when it is going to be used it must be copied from where it is saved to the special memory region. This is done using the following methods:

```c
bool commitVertexProgram( u32bit memDesc, u32bit programSize, u32bit startPC );

bool commitFragmentProgram( u32bit memDesc, u32bit programSize, u32bit startPC );
```

These functions are used to copy the shader program code from the GPU memory region where it is saved to the special memory regions from where shaders can be executed. The parameters of these functions are the MD where the shader is saved in the GPU ‘memDesc’, the size of the program ‘programSize’ and which is the first instruction to execute ‘startPC’.

All these are the functions the Driver exports to the upper levels. They are quite simple but allow us to use the memory of the GPU. Now the problem is how the ACD maintains synchronized the resources between the API and the GPU.

Next step is explaining how the MemoryObjectAllocator works. This object is the responsible for tracking and maintaining synchronized all the resources that are going to be used and there is only one allocator for all the system. When the MemoryObject was explained nothing was said about where each region is going to be saved in the GPU, there is no map between the each memory region in the MemoryObject and the MD where the data is saved. This information is saved by the MemoryObjectAllocator which saves the mapping between every region allocated in the system and the memory descriptor. This may seem a bit contradictory, because it is more logic to save this information inside the struct where the region state is saved. This is done this way because memory management is very important and it is desired that some element of the system can easily apply different allocating polices. As MemoryObjectAllocator have all this information, it is easy to it to manage the whole GPU memory, and for example reallocate the whole memory to optimize it. If MDs are saved in each MemoryObject, every time the state of the GPU memory would like to be analyzed, all the MemoryObjects of the system must be traversed.

Then, the first task of the MemoryObjectAllocator is mapping each region of the MemoryObject with its MDs; remember that some MemoryObjects like the ACDTexture have more than one region. This task takes place when the memory region is synchronized for the first time.

Apart from this, the MemoryObjectAllocator main capability is to synchronize the data with the GPU. This is done using the following methods:

```c
void syncGPU(MemoryObject* mo);
void syncGPU(MemoryObject* mo, acd_uint region);
```
The first one is used to synchronize all the regions of a MemoryObject, and the input parameter is the MemoryObject ‘mo’ that contains all the regions. The second method is used to synchronize a specific region of one MemoryObject. Then, the input parameters are the MemoryObject ‘mo’ and the number of the region ‘region’, to synchronize.

The way how the syncGPU function works is quite simple. For each region, in the first function, or for only one region in the second, it obtains which is the state of the region from the MemoryObject class. Then, depending on the state in which the region is, the job of the MemoryObjectAllocator is to synchronize the region if it is needed.

In case the region is MOS_Sync nothing has to be done as the region is already synchronized. When the state is MOS_Realloc, it is necessary to reallocate the memory region. This situation can take place in two different scenarios, when is the first time the region is allocated, then there is no map between the region and a MD, so a new memory space must be allocated and mapped to the region, or when exists a MD for the memory region but where the resource data doesn’t fit in which case the old MD must be deallocate and a new MD obtained.

MOS_NotSync state takes place when the resource data have been updated but its size has not been increased. In this case a map between the region and the MD description exists and only the memory region in the GPU must be updated. To reduce the traffic of the AGP bus, instead of resynchronizing the whole region the MemoryObjectAllocator use the ‘firstByteToUpdate’ and ‘lastByteToUpdate’ variables that have been seen in the MemoryObject to avoid updating unnecessary data.

When the state of a region is MOS_Blit or MOS_RenderBuffer the MemoryObjectAllocator doesn’t do anything as the newest version of the data is saved in the GPU.

Finally the MemoryObjectAllocator provide two functions used to deallocate resources from the GPU.

```c
acd_bool deallocate(MemoryObject* mo);
acd_bool deallocate(MemoryObject* mo, acd_uint region);
```

They are mainly used when a resource is deleted from the API, and before this is done, the GPU memory must be freed. The parameters of these functions are the same as others seen.

### State synchronization

Another important thing that must be synchronized is the ACD state. The ACD saves the state inside some structures, and these structures must be synchronized with the GPU. These structures are modified using the methods described in the previous chapter. This state must be propagated to the GPU, but another time this propagation must be done in an appropriate way to avoid extra AGP transactions that increase the bus traffic.
The state of the GPU is saved in some GPU registers. There is more than a hundred registers and explaining all of them is not necessary to understand the whole project. The driver provides the `void writeGPURegister( GPURegister regId, GPURegData data)` to write the GPU registers and update the GPU state. The `regId` value is the name of the register that is being write and the `data` value the data set into this register.

Synchronization seems something easy, but in such a large structure such as the ACD it is not so easy. In the ACD the state is not centralized in one element, it is distributed among different elements; this difficulties how it is synchronized.

The state is distributed between the ACDDDevice which contains most of them, each ACDSStream where there is the state of every stream, the ACDRasterizer where is the state of the raster unit, each ACDSampler where there is the state of every sampler and finally the ACDBlendingStage where blending is configured.

First important question to answer is when the ACD state is going to be synchronized. There are two main possibilities, do it every time the state changes, then every time the API user modifies the state, or do it only when it is necessary, when a draw call is issued. Obviously the answer is that it must be done only when a draw call is issued. The reason is because many APIs modifies several times one parameter before issuing a draw call, being only valid the last value set. Imagine that every time this parameter is changed the GPU state is changed; this creates AGP transactions that don’t do useful work, only the last state change is useful.

Then state synchronization takes places every draw call. An extra problem appears as state is distributed among different object, and all of them must be synchronized, and the problem is bigger if this has to be done by the programmer who is using the ACD. To avoid that the user has to synchronize every object that the state has, this is done automatically by the ACD when a draw call is issued. As all draw calls are inside the ACDDDevice this class is the responsible of synchronizing the whole state of the ACD.

Every object that contain a piece of the state has a synchronize function. In these functions the appropriate GPU registers are synchronized. Then the ACDDDevice performs the whole synchronization, synchronizing the piece it has of the state and issuing a synchronize call to every other element.
Another topic to discuss is when the resources are synchronized with the GPU. The most logical point to place this process is inside the sync() function of the object that contains the resource. Then, apart from synchronizing the state, each object that contain any resource (textures, shaders or buffers) must call the appropriate syncGPU function in the MemoryObjectAllocator to synchronize the resources.

**Optimization to reduce the traffic of the AGP bus**

Last section of this chapter analyzes which structures are used by the ACD to minimize the bus overhead and be more accurate. Some of these techniques have been explained before.

The MemoryObject class, apart from saving in which state a memory region is, it also saves the range of bytes that have been updated. This is useful as usually not the whole region is updated, so with this information it is possible to reduce the amount of data transferred when a region is updated.

Another way to reduce the traffic of the bus is reduce the number of AGP transactions. Every time a GPU register is written it is necessary that the ACD issue an AGP transaction, so the best way to minimize the number of AGP transactions is minimizing the number of state changes.

This has been done when it has been decided to only synchronize the state when a draw call is issue, but there is another problem. Even if the state is only synchronized every draw call, many of the registers may not have changed their values, or they may have changed by the same value. These changes are not really need to be synchronized as they are already synchronized, so it is necessary to avoid them to reduce the number of AGP transactions.
The way to do this is track every value of the state, and when the sync() function is called synchronize only the registers that have been modified. To do this the ACD has added a new class called StateItem. StateItem class is a container for whatever variable type that tracks if the value of the variable has been changed. Then it is easy to know if the variable has changed and in that case synchronizes it. StateItem structure saves the original value that was set and the current value and then it only needs to compare both values to know if it has been changed. The best way to understand how the StateItem works and show the different functions it has is explaining an example.

It is desired to track the value of an integer. Then a Stateltem of type integer is created.

```cpp
StateItem<acd_uint> example (0 [initial value]);
```

Now the original and current value is set to 0
With this variable whatever operation can be done.

```cpp
Example = 1;
Example = Example * 2;
```

All these operation modify the current value, but the original value continues being the same, 0. Then to know if the original value has been modified:

```cpp
Example.changed()
```

is used. This function returns if the original value has changed, and in the example it does because after the previous sequence of operations the current value is 2 which is different from the original value, 0. So the result is true.

After checking the state original, the value continues being 0, as the changed() operation doesn’t change it. To update the original value with the current value this function must be used:

```cpp
Example.restart();
```

Now, original value is 1.
Finally there is another function that allows knowing which has been the original value of the Stateltem. This function is:

```cpp
Example.initial()
```

which returns a 0.

The Stateltem element is very useful to track if the different parameters of the state have changed. Then, the Stateltem object is used for every parameter that saves the state. When the
different parameters of the state are synchronized, the first thing to do before writing the GPU register is find out if the register has changed from the previous value. This is done using the changed() function from the Stateltem. If this has happen, then the register is written and the Stateltem is synchronized with the restart function. If it has not been modified, nothing is done. This technique avoid writing unnecessary registers.
This chapter explains some details about how the ACD is implemented internally. Previous chapters have shown how big the ACD structure is. Even being so big, from the point of view of the programmer, it is not very difficult to implement as it doesn’t contain complex algorithms. This chapter only explains the most interesting topics about the ACD implementation, assuming that the simplest ones are not interesting from the point of view of an engineer.

The first important part is analyzing which are the steps the ACD do to issue a draw call. It is interesting because this operation is the responsible for synchronizing all the state and the resources set in the ACD.

Apart from this, the most interesting part of the ACD is when it has to implement capabilities that are not supported by the underlying hardware. In this case, it is necessary to deeply understand which are the capabilities the hardware offers and the requirements of the function to implement. Then, it is necessary to mimic using the available hardware the behavior of the function.

Another interesting part is the texture compressor. The reason to implementing it is because some API provides the capability to compress a texture from a non-compress format.

Finally, it is necessary to talk about how the ACDRenderTarget objects are created from an ACDTexture object and also how the clear operations are programmed.

**Draw Calls**
One of the most important parts of the ACD is how the state and the resources are synchronized with the GPU. The previous chapter has explained which is the structure followed to improve the way the synchronization take place. This section completes the information provided there explaining how it is done.

To explain this topic it is used the code from the main draw operation in the ACDDevice, not explaining in details which is the code of each specific synchronization function. The reason for this decision is very logical because how each parameter is synchronized have been deeply explained in previous chapters.

Before starting it is important to understand that the order in which the different synchronization operations take place don’t have to be necessarily the order of the graphic pipeline. There reason is quite simple, it doesn’t matter in which order the registers are synchronized, what it is important is that at the end everything is set appropriately.

Then, to follow the explanation the source code of the draw call function is used.

```c
void draw(acd_uint start, acd_uint count, acd_uint min, acd_uint max,
        acd_uint baseVertexIndex, acd_uint instances)
{
    The first thing is synchronizing the render targets. This function checks for every render target slot if the ACDRenderTarget object set to it has changed. In case any render target has changed it is also necessary to flush the content of the color caches of the GPU because they save information about the render target. To flush the color caches the Driver offers the following function _driver->sendCommand(gpu3d::GPU_RESET_COLOR_STATE).

    _syncRenderTarget();

    The next thing is synchronizing the primitive type. As this information is saved inside the ACDDevice this is done inside the draw function. This is a good example to explain how a parameter of the state is synchronized with the GPU using the StateItem class methods. The variable being synchronized is '_primitive', and the first thing is checking, using the changed() function, if the value have changed from the previous time the state has been synchronized. If it hasn’t changed anything have to be done, on the other side if it has changed the new value have to be synchronized with the GPU. This is done writing the appropriate GPU register with the new value. Remark that the '_translatePrimitive' method is used to translate from the ACD type to de GPU type. These types of functions are very common as the types of the GPU and the ACD are different. To finish with the synchronization it is need to restart the value of the variable to the new one.

    if ( _primitive.changed() )
    {

```
gpu3d::GPURegData data;
_translatePrimitive(_primitive, &data);
_driver->writeGPURegister(gpu3d::GPU_PRIMITIVE, data);
_primitive.restart();
}

Next task is synchronizing the ACDRasterizationStage, the ACDZStencilStage and the ACDBlendingStage object. All of them are quite simple as they don’t synchronize any resource, only the state. Then, for each parameter of the state they check if it has changed, using the same method described above.

_rast->sync();
_zStencil->sync();
_blending->sync();

The synchronization of the shaders can be divided in two different parts, the shader code and the constant table synchronization. How both are synchronized is different because the shader code is synchronized as a resource, so using the appropriate syncGPU function in the MemoryObjectAllocator, while the constant table is synchronized as other variables that save the state.

It is important to remember that, apart from having to synchronize the shader code it must be copied to the memory region reserved to execute the shader code. This is done using the appropriate commit*Program method provided by the driver.

_syncVertexShader();
_syncFragmentShader();

The ACDSampler is the class that contains the information to configure the texture units. All the ACDSampler objects are saved inside the ACDDevice, and what the '_syncSamplerState' function does is issuing the sync() function for each ACDSampler object.

Inside the sync() function the first thing to consider is if this sampler is enabled, because if not there is no need in continuing synchronizing the rest of the registers as they don’t do nothing. Working this way reduces the number of written registers.

Enabling and disabling the samplers is done by the user, but sometimes they don’t do this appropriately. This is not done appropriately when the sampler is enabled and it is not used by the fragment shader. To improve the performance what is done is disable those samplers that are enabled but not used inside the fragment shader.

The synchronization of the ACDSampler object has different steps. The first one is synchronizing the parameters saved inside the ACDSampler objects, which basically are the filters, LOD levels, etc. Next step is synchronizing the ACDTexture object set to this sampler.
The ACDTexture contains some characteristics that must be synchronized with the GPU (minimum and maximum mipmap…) and also the set of Mipmap objects. Each Mipmap contains all its characteristics (width, height,…) as well as the mipmap data. As the mipmap data is a resource it is synchronized using the syncGPU function in the MemoryObjectAllocator.

_syncSamplerState();

The way how the ACDStream is synchronized is very similar to the ACDSampler. The ACDDevice also contains all the ACDStreams available. Each ACDStreams synchronize the information of the stream (stride, offset, type…) and the ACDBuffer content using the MemoryObjectAllocator. This is done only in case the ACDStream is enabled, otherwise not. Also it is possible to disable the ACDStream that are not used inside the vertex shader program, as it has been done with the ACDSamplers.

_syncStreamerState();

Other important information is how many vertices are read from the streams. This information is set when the draw call is issued. The parameters for draw calls have been analyzed in the previous chapter.

if (_indexedMode && baseVertexIndex != 0)
{
    _addBaseVertexIndex(baseVertexIndex, start, count);
    start = 0;
}
.syncStreamingMode(start, count, instances, min, max);

Apart from the Z and the Stencil test configured before, it is necessary to configure other tests to discard fragments. Two of them are the hierarchical Z test and the earlyZ test. Remember that both tests are managed internally by the ACD and not by the user of the ACD.

The hierarchical Z test can only be used in certain situations:

- Z test must be enabled
- Z function must be ACD_COMPARE_FUNCTION_LESS or ACD_COMPARE_FUNCTION_LESS_EQUAL or ACD_COMPARE_FUNCTION_EQUAL
- If Stencil buffer is active stencil operation must be ACD_STENCIL_OP_KEEP when the stencil test fails and when the stencil test passes but the depth fails.

If all these conditions are met, the Hierarchical Z test can be enabled, otherwise not.

_syncHZRegister();
Apart from Hierarchical Z, the ACD also configures in which situations the Early Z test can or not be enabled. From the previous code it can be seen that early Z can be active if the shader don’t have any kill instruction and fragment shader output number 0 is not active. If these two requirements are not met early Z must be disabled.

The early Z test can only be activated if the fragment shader doesn’t have any kill instruction and the fragment output 0 is not active. If these to requirements are met the early Z must be disabled, otherwise it can be enabled. The following code manages the early Z test.

```cpp
ACDShaderProgramImp* fsh = _fsh;

if (fsh->getOutputWritten(0) || fsh->getKillInstructions())
{
    _earlyZ = false;
    if (_earlyZ.changed())
    {
        bValue.booleanVal = false;
        _driver->writeGPURegister(gpu3d::GPU_EARLYZ, bValue);
        _alphaTest.restart();
        _earlyZ.restart();
    }
}
else
{
    _earlyZ = true;
    if (_earlyZ.changed())
    {
        bValue.booleanVal = true;
        _driver->writeGPURegister(gpu3d::GPU_EARLYZ, bValue);
        _alphaTest.restart();
        _earlyZ.restart();
    }
}
```

All the operations described above have configured the entire GPU. Now to draw the batch using this configuration the following command have to be send to the GPU.

```cpp
_driver->sendCommand( gpu3d::GPU_DRAW );
_currentBatch++;
```

**Texture compressor**

Some APIs provide the capability to compress a non-compress texture. This is not the typical function an API has as this task is usually done by the artist who designs the texture.
This functionality is provided by the ACD using a texture compressor. There are many compressed formats and implementing all of them is a hard work and useless. Then available compressed formats are implemented on demand. Another problem with the compression formats is that it is not provided the algorithm to compress a texture into a given compressed format. The only algorithm provided is to decompress the texture. So every compressor has to design and implement its own algorithm.

By the time the ACD texture compressor only allows compressing from the RGB and RGBA formats to the S3TC_DXT1 format which is the only one required to support the workload used for this project. So, before continuing explaining how the compressor is implemented, let’s take a look how the S3TC_DXT1 decompression algorithm is. Remember that this is the only information available.

Compressed texture images stored using the S3TC_DXT1 format are saved using blocks, where each block contains 64 bits of data and represents 4x4 texels of the original texture.

Then a S3TC_DXT1 image with a width of ‘w’, height of ‘h’, and 8 bytes of block size is decoded, the corresponding image size (in bytes) is:

\[
\text{imageSize} = \left\lceil \frac{w}{4} \right\rceil \times \left\lceil \frac{h}{4} \right\rceil \times 8
\]

As the S3TC_DXT1 format saves the texture texels in blocks the most important information to know is given a texel from the original image (‘x’, ‘y’) in which block the data is found. This formula:

\[
\text{offset} = 8 \times \left\lceil \frac{w}{4} \right\rceil \times \left\lfloor \frac{y}{4} \right\rfloor + \left\lfloor \frac{x}{4} \right\rfloor
\]

compute the offset (in bytes) to the block which contains the desired texel.

Each RGB image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

\[
c0_{\text{lo}} \ c0_{\text{hi}} \ c1_{\text{lo}} \ c1_{\text{hi}} \ \text{bits}_0 \ \text{bits}_1 \ \text{bits}_2 \ \text{bits}_3
\]

The 8 bytes of the block are decoded into three quantities:

\[
\text{color}_0 = c0_{\text{lo}} + c0_{\text{hi}} \times 256 \\
\text{color}_1 = c1_{\text{lo}} + c1_{\text{hi}} \times 256 \\
\text{bits} = \text{bits}_0 + 256 \times (\text{bits}_1 + 256 \times (\text{bits}_2 + 256 \times \text{bits}_3))
\]

And from this information the colors of the 16 texels inside the block are obtained.
The color0 and color1 quantities are two 16-bit unsigned integers that are unpacked to two RGB colors, RGB0 and RGB1, with a type of UNSIGNED_SHORT_5_6_5.

Bits is a 32-bit unsigned integer, from which a two-bit control code is extracted for each texel in the block. Fromm the following formula, being ‘x’ and ‘y’ the coordinates of the texel inside the block the 2 bit code is extracted:

\[
\text{code}(x,y) = \text{bits}[2^*(4*y+x)+1..2^*(4*y+x)+0]
\]

Then using the code and the two RGB colors, the RGB color for a texel at location \((x,y)\) in the block is given by:

<table>
<thead>
<tr>
<th>RGB0,</th>
<th>if color0 &gt; color1 and code(x,y) == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB1,</td>
<td>if color0 &gt; color1 and code(x,y) == 1</td>
</tr>
<tr>
<td>((2\times\text{RGB0}+\text{RGB1})/3),</td>
<td>if color0 &gt; color1 and code(x,y) == 2</td>
</tr>
<tr>
<td>((\text{RGB0}+2\times\text{RGB1})/3),</td>
<td>if color0 &gt; color1 and code(x,y) == 3</td>
</tr>
<tr>
<td>RGB0,</td>
<td>if color0 &lt;= color1 and code(x,y) == 0</td>
</tr>
<tr>
<td>RGB1,</td>
<td>if color0 &lt;= color1 and code(x,y) == 1</td>
</tr>
<tr>
<td>((\text{RGB0}+\text{RGB1})/2),</td>
<td>if color0 &lt;= color1 and code(x,y) == 2</td>
</tr>
<tr>
<td>BLACK,</td>
<td>if color0 &lt;= color1 and code(x,y) == 3</td>
</tr>
</tbody>
</table>

Arithmetic operations are done per component, and BLACK refers to an RGB color where red, green, and blue are all zero.

Since this image has an RGB format, there is no alpha component and the image is considered fully opaque.

This is an example of the explanation:
With this compression algorithm a 32:1 compression rate can be obtained.

The main important parts of this algorithm are that the texture resource is divided in blocks of 4x4 texels. Each of these blocks has to be regenerated from two colors of type \texttt{UNSIGNED\_SHORT\_5\_6\_5} and 2 bits for each texel. These bits are used to determine which combine operation is going to be used to generate the texel color from the two defined color of this block.

Then, the work of the compression algorithm is find which are the best two colors that, using the available combine operations, generate the most similar colors for the texels of the block. There are many ways how this can be done, and to obtain the best quality color theory must be studied.

The objective of the texture compressor of the ACD is not to obtain the best quality, only obtain something similar. The Attila stack is design to simulate a real GPU architecture, how well a texture is compress doesn’t change anything in the GPU architecture, so this compressor doesn’t have to be the best.

The first approach to this problem is using a brute-force algorithm. The problem is to find the best two colors from which, using the combine operation, the colors obtained for the entire block (when they are decompressed) are the most similar to the original ones. There are infinite possible colors to combine, but taking into consideration that the colors of the texels inside the block are quite similar, the best way to find the best two colors is using colors from the block. Then, what this algorithm does is for each pair of texel colors inside the block it finds the best combination function for each texel that obtains the most similar color to the original one.
result of this algorithm is the two best colors from which, using the available combination functions, the best texel colors are obtained. It is important to highlight that for us the best color is this that the difference between the original color and the color obtained from the decompression is the smallest, being the smallest when the subtraction of each component is the smallest.

This algorithm has some problems. The first one is that it takes too much to evaluate all the possible combinations. This problem is bigger as new games use bigger textures. Another problem is that the colors selected for the block can only be two of those that are in the block, and may be exists 2 more colors that generate better colors for the block. Finally, how the difference between two colors is defined is not as accurate as it has been defined. Computing the difference between two colors is something much more difficult and needs a deep understanding of the color theory.

From the purpose of the algorithm the results are quite good, but the main problem is that it requires too much time to compress the textures. To reduce the amount of time it takes instead of choosing which are the best two colors of the block, they are selected randomly. With this solution the algorithm takes much less because the algorithm is reduced to find the best combination functions for each texel but using always the same two colors. The results have practically the same quality as the previous ones, so it has been decided to use this algorithm.

Figure 44: Original image without using the texture compressor
Figure 45: Original image where all the textures have been compressed using the texture compressor provided by the ACD.

Figure 46: Difference between the original image using non-compressed texture formats, and the image using compressed texture formats. The PSNR (Peak Signal-to-Noise Ratio) is 40.8092 which is a very good result.
Clear operations

Another interesting topic is how the ACD implements the clear operations. Remember that these operations are used to clear one render target, the stencil or Z buffer with a given value.

Clearing the whole surface of whichever of these resources can be done easily issuing the appropriate GPU command, which are `GPU_Z_BUFFER_CLEAR` to clear the Z buffer, `GPU_STENCIL_BUFFER_CLEAR` to clear the stencil buffer, `GPU_CLEARZSTENCILBUFFER` to clear both, the stencil and the Z buffer, and finally the `GPU_CLEARCOLORBUFFER` to clear the render target set at the first slot.

The problem appears when it is desired to clear only a region of one of these resources. GPU architecture doesn’t provide any operation to do this task, so it is necessary to emulate it. This type of clear is called partial clear.

The basic idea to do a partial clear is issue a draw call which main geometry is a square (two triangles) with the same area that the area wanted to clear. Then setting appropriately the state of the GPU, it is possible to obtain the results expected.

Let’s analyze each of the different clears how can be obtained.

When the Z Buffer is partially clear the rest of the buffers don’t have to be modified. To avoid this, the color write to all the frame buffers is disabled, so nothing is written in it. Also, to avoid modifying the Stencil buffer, the stencil test is disabled. Now the problem is set the appropriate value to the cleared area. To do this Z test function is going to be set to `ACD_COMPARE_FUNCTION_ALWAYS`. Then all the fragments of the square would pass the Z test and would be written in the Z Buffer. Then, it is necessary to set the appropriate Z value to each vertex of the geometry in such a way that the fragments generated in the rasterization have the desired clear value. This value would be written in the Z buffer, obtaining the same result as a clear operation.

The partial clear for the stencil buffer is done more or less like the Z one. The main difference is that this time the color write and the Z test are disabled to avoid modifying them. Now the stencil test is enabled. Then, the stencil function is set to `ACD_COMPARE_FUNCTION_ALWAYS`, allowing that all the fragments of the square pass the stencil test and then the stencil operation applied if the stencil test is passed is set to `ACD_STENCIL_OP_REPLACE`. With this function every time one fragment passes the stencil test, the stencil buffer is update with the reference value. So, the last thing to do is set the clear value as the reference value.

It is possible to do a Z and Stencil partial clear together. This time the state must be configured using both configurations, and that’s all.
Finally, last partial clear is done on the main render target. This time the Z and the stencil tests are disabled and color write enabled. Now, every the color of each vertex of the square must be set using the clear color. Nothing more has to be done; the fragments generated from the rasterization of the geometry clear the desired area using the color set to the vertices.

It is important to highlight that to do a partial clear it is necessary to modify the state of the ACD. These modifications, after the partial clear has been done, have to be reverted to the previous state.

**Create a RenderTarget**

As previously seen, an ACDRenderTarget is created from a mipmap level of an ACDTexture. As Attila uses a different internal layout for the render target and the texture data, it is not possible to work with both at once in a simple way. Another problem is that the render target objects are modified inside the GPU, so the information in the API is not synchronized with the GPU. The problem here doesn't have any solution, as the Attila architecture doesn't provide any read operation that allows reading the render target to update the data inside the API. So, once a render target is created from one texture, it is not possible to update the texture content, only use the texture.

This section focuses on which are the steps necessary to transform to update the current texture mipmap, which cannot be used as a render target, to a resource capable of being a render target.

The first important thing to think about is that the texture can be probably synchronized with the GPU. If this happens, this information is not useful as it is saved using the Morton layout. So first step is deallocating the memory area the texture mipmap is using. It doesn't matter that the data is removed as the data is also saved in the API.

Next step is allocating a memory region to save the render target. As the render layout is a bit different from previous seen, the Driver provides a function called createRenderBuffer(u32bit width, u32bit height, bool multisampling, u32bit samples, gpu3d::TextureFormat format) to obtain a memory region with the required size to save the render target. The input parameters are the 'width' and the 'height' of the texture, if it is using 'multisampling', the number of 'samples' and the 'texture format'. This function returns the MD of the memory region.

Next step is binding the MemoryObject where the mipmap region is saved with this new MD. This is done using the MemoryObjectAllocator function void assign(MemoryObject *mo,
acd_uint region, acd_uint md), where ‘mo’ is the MemoryObject where the mipmap region is saved, ‘region’ is the mipmap level desired to configure and ‘md’ is the new MD to the ‘region’. Now the new MD is bind with the mipmap.

Next step is transforming the texture data to the render target layout. This is also done using a function provided by the driver, `void tileRenderBufferData(u8bit *sourceData, u32bit width, u32bit height, bool multisampling, u32bit samples, gpu3d::TextureFormat format, u8bit* &destData, u32bit &size)`. The necessary information from the texture data is a pointer to the data ‘sourceData’, the ‘width’, the ‘height’, if uses ‘multisampling’, the number of ‘samples’ and the ‘textureFormat’. ‘destData’ and ‘size’ are output parameters that return the texture data using the render target layout and the size of this data.

Once this is done, is time for synchronizing this data to the memory region previously obtained using the syncGPU function.

Finally, as the layout of the resource have changed, it is necessary to change it in the ACDResource class setting it to ACD_LAYOUT_RENDERBUFFER.

From now on, this texture mipmap is a render target and cannot be updated anymore by the API. Of course it can be used also as a texture, but taking into consideration that it uses the render target layout, and this must be set in the GPU when it is used as a texture.

**Copy operations**

The final topic of this section explains how the copy operations are performed. The available copy operations are between two mipmaps, between a mipmap and a render target and between a render target and a mipmap.

Copy one mipmap to another mipmap is done in different ways depending on the source and the destiny areas.

In case the destiny and the source areas have the same size this operation is very easy because it is not necessary to apply any filter and also because the data of the mipmaps is saved inside the API. Then, what it is done is obtaining the data from the source mipmap and copying it to the destiny mipmap using the getData and setData operations provided by the mipmap class and explained in previous chapters.

In case the source and the destiny areas have a different size it is need to use a difficult mechanism similar to those used with the clear operations. The idea is use another time is rendering a square, with the same area as the destiny area, on the destiny mipmap. This time each vertex of the square have the appropriate texture coordinates to map on the square the source area of the mipmap. Also it is necessary to configure the filters requested by the user in the texture unit used. To be able to use this technique it is necessary to turn the destiny mipmap
into a render target to be able to render on it the square. The consequences of this using this technique is that the texture cannot be updated anymore, which can be a problem if the API updated the mipmap, but by the time this hasn’t happened.

Next available operation is copy a render target to a mipmap. Remember that in this case the source and the destiny region have to have the same size. This operation is very simple as it is done using the blitter unit of the hardware. This unit is configured using some registers and issuing the GPU_B_\text{LIT} command.

Finally it is available the operation to copy from a mipmap to a render target. The technique used is practically the same used when it is necessary to copy between two different mipmap with different regions. In this case the source is a render target, there is no problem in using it as a texture because Attila supports using the render target layout as a render target. Another time the destiny is a texture so it is necessary to turn it into a render target, having the drawbacks previously explain. Apart from that everything is done exactly as explained before.
The previous chapters have described the ACD from the point of view of the programmer who is going to use the interface and from the point of view of ACD programmer.

Another objective of this project is to validate that the ACD design works properly and test it using real workload. Using real workload directly on the ACD is impossible as there is any application that uses the ACD interface. The only way to do this is implementing one specific API on top of the ACD. To this task it has been decided to implement the openGL library.

This chapter describes how the AOGL (Attila OpenGL) has been designed. Implementation topics are practically not covered as the AOGL doesn’t have any difficult algorithm.

**Overview**

OpenGL architecture is a state machine. It is important to take into consideration this fact as the ACD doesn’t work as a state machine, so the AOGL layer must mimic the openGL state machine behavior to set the ACD state appropriately. Also, the AOGL must expose all the available openGL calls in such a way that allows the Player, which is on top of the AOGL and read the trace file, to call them easily.

The following schema shows how the AOGL is designed.
The first element the AOGL has is the AGLEntryPoint. The AGLEntryPoint exposes all the openGL functions implemented in the AOGL. The AGLEntryPoint is connected to the openGL Player which reads the trace and issues the appropriate function inside the AGLEntryPoint for each call read.

The most important element the AOGL has is the AGLContext. The main challenge of this element is manage the openGL state machine, so it must emulate the openGL state machine to set appropriately the ACD. Then the AGLContext object is responsible for translating the openGL state to the ACD. As the ACD doesn’t support fixed function, the AGLContext is connected to the ACDX that translated fixed functions configurations into the appropriate shaders and also compiles the ARB code into the Attila Byte Code.

The AGLEntryPoint and the AGLState are connected, so for each implemented openGL call the AGLEntryPoint object must set the appropriate state inside the AGLContext object. As there are some parameters that aren’t part of the AGLContext, the AGLEntryPoint is also connected with the ACD, and those parameters are directly set in the ACD.

Finally to manage the available openGL resources, AOGL has some structures called AGLBaseManager, AGLBaseTarget and AGLBaseObject from which each specific resource inherit and have its own version.

The following sections explain in more detail these elements.

**AGLEntryPoint**

The AGLEntryPoint is the AOGL interface exposed to the above levels. This time the interface used for the AOGL is the openGL interface, as the AOGL is the layer that translates the openGL calls into ACD calls.
OpenGL have hundreds of calls, but, as previously seen, AOGL doesn't support all of them, only a small subset. Calls in this subset are those required to run the game, no more. Something peculiar to this subset is that it doesn't include any getter. Programmers use them to obtain certain information about the openGL state. As our workload is saved in trace files, so it is previously recorded, the sequence of instructions is already determined, so these getter calls are not useful as their answer won't change how the trace works. That's the reason why these calls have not been implemented.

As the AOGL mimics the openGL interface, the AGLEntryPoint implemented calls must provide the same interface as the openGL call. Then, it basically contains one function for each implemented call which has the same interface as the openGL one.

Here are some examples:

```c
GLAPI void GLAPIENTRY AGL_glFrontFace( GLenum mode )
GLAPI void GLAPIENTRY AGL_glFrontFace( GLenum mode )
GLAPI void GLAPIENTRY AGL_glDepthMask( GLboolean flag )
GLAPI void GLAPIENTRY AGL_glEnableVertexAttribArrayARB (GLuint index)
```

These functions are not too complex as they only set the appropriate state to the AGLContext object. The way the state is set is very easy because the AGLContext object tries to provide simple functions to avoid that the AGLEntryPoint functions has to do complex thing to set the state. Then each of these functions is basically a sequence of calls issued to the AGLContext which mainly set it.

This is an example of how an openGL function set the appropriate state in the AGLContext object (the '_ctx' variable is the AGLContext object). In this case this call is used to draw a vertex.

```c
GLAPI void GLAPIENTRY AGL_glVertex3fv( const GLfloat *v )
{
    _ctx->addVertex(v[0], v[1], v[2]);
}
```

Some parts of the AGLState don't need to be maintained in the AGLContext object. Then, these elements can be set directly in the ACDDevice. Here you can find an example of how the viewport size is set.

```c
GLAPI void GLAPIENTRY AGL_glViewport( GLint x, GLint y, GLsizei width, GLsizei height )
{
    ACDDevice& acddev = _ctx->acd();
    acddev.rast().setViewport(x, y, width, height);
}
```
Finally there are some functions that change the state of the available resources (textures, shaders or buffers). This part is done using each available AGLBaseManager which provides access to the stored resources. This part is a little more complex as it requires explaining how the AGLBaseManager, the AGLBaseObject and the AGLBaseTarget work. So this part is converted more in detail when the internal AOGL resources are explained.

Here you can find an example of a function that changes the AGLBufferObject currently bind.

```c
GLAPI void GLAPIENTRY AGL_glBindProgramARB (GLenum target, GLuint pid)
{
    ARBProgramManager& arbPM = _ctx->arbProgramManager();

    if ( pid == 0 )
        arbPM.target(target).setDefaultAsCurrent();
    else
    {
        // Get/Create the program with pid 'pid'
        ARBProgramObject& arbp =
            static_cast<ARBProgramObject&>(arbPM.bindObject(target, pid));

        arbp.attachDevice(&_ctx->acd());
    }
}
```

**AGLContext**

The AGLContext is the most complex element the AOGL has. It has to manage the whole openGL state, which include the resources. Also, the AGLContext has to deal with the ACDX unit to create shaders that emulate the fixed function state because the ACD layer doesn’t support it and also compile the ARB code to the Attila Byte Code.

As it has been said, openGL is a state machine, so when it is desired to modify a resource, the first thing that must be done is set this resource in the state machine. Once the resource is set it is possible to issue operations that modify it. The difference from other paradigms is that here the user doesn’t work with the object, it only works with a state machine and all the operations issued are on this state machine.

Then, one of the tasks of the AGLContext is save which is the current openGL state. This is very important because the following operations done modify this state.

Another important task of the AGLContext is mapping the openGL types with those from the ACD. Usually this mapping is very easy as it is only translating the openGL syntax to the ACD one, but sometimes this is more complex because the ACD doesn’t provide an equivalent structure to the openGL one.
One of the most complex tasks is mapping how openGL declares the vertices to the way the ACD does. Remember that openGL provides 3 different ways to define the vertices. Using the `glBegin()/glEnd()` structure, with the Vertex Buffer Objects (VBO) or with the Vertex Arrays. How all of them work has been explained in the second chapter.

In the `glBegin()/glEnd()` structure, each vertex is defined one by one, and this is a problem as the ACD only supports vertex buffers using the ACDBuffer object. To emulate this behavior the AGLContext creates as many ACDBuffers as vertex attributes are enabled. Each one is bind to one of the vertex attributes. Then, every time a new vertex is added, each current attribute value is added to each vertex buffer. In the end, these ACDBuffers are filled with all the information necessary to render the scene. The problem with these ACDBuffers is that they cannot be used in the next draw call, so they must be deallocated from the GPU in order to save space.

Vertex array can be map directly to the ACDBuffers as both are vertices that contain vertex attributes. The problem with them is that they are saved in the user space. So the user can modify them without notifying the API. Then it is not possible to trust that between two draw calls the same pointer refers to the same data. Because of this problem, the Vertex Arrays cannot be reused, and as it happens with the `glBegin()/glEnd()` structure they must be deallocated after the draw call.

Finally, as with the vertex arrays, the VBO can be map directly on an ACDBuffer object. The main advantage this time is that the data is saved in the API memory space, so whatever modification make on the VBO have to be done throw the API. As the API tracks all the modifications done on the ACDBuffers it is possible to reuse the buffer between different draw calls. Then, the next time it is used, if it has not been modified, the ACD doesn’t have to synchronize the ACDBuffer another time as it is already inside the GPU. Also, when the buffer is updated, there is no need to destroy the entire ACDBuffer as the update operation is done throw the API, so it can track it.

Another problem is that openGL doesn’t support streams, so there isn’t any map to the ACDStream unit. Then, it is necessary a way to assign and distribute the ACDStreams and this is done using a pool of streams. In every draw call, every vertex attribute enabled request a ACDStream to the pool, use it and after the draw call the ACDStream is released.

Configuring the texture units, or other stages such as blending, the Z test, the Stencil test and other characteristics is much more simpler as the map between openGL and DirectX is immediate.
ACDX

The ACD doesn’t provide support for fixed function. To solve this problem it is necessary to generate shaders that emulate the behavior of the fixed function part. Remember that the fixed function part configures the transform and lighting stages (T&L). This generator is included inside the ACDX layer. As this is a very complex part that requires a deep knowledge of shader programming and fixed function configurations it has been done in another project. More references can be found in (Roca Monfort, 2005)

This section doesn’t explain how the shader generator is internally programmed, only the how an external user, in this case the AOGL, can configure this layer to use the shader generator.

The ACDX interface provides two different objects to configure the shader generator: the ACDXFixedPipelineState and the ACDX_FIXED_PIPELINE_SETTING. Both structures are used to configure the transform and lighting state, and it is important to know what are the differences between them. The ACDX_FIXED_PIPELINE_SETTING is a big structure used to enable, disable and characterize the different elements that belong to the transform and lighting stages. An example of what can be done here is enabling a light and selecting a spot light type. The ACDXFixedPipelineState is used to configure the properties of the different elements, such as the color of the light, the cut-off angle, etc. The reason why this is done this way is because the first structure contains all the parameters that modify how the shader is built, depending the number and type of lights enabled the shader is different, while the second structure contains parameters that don’t modify the shader, such as the color of the light, which is saved as a constant.

After each structure has been configured appropriately is time to generate the shaders from these configurations. This is done using one of the following functions provided by the ACDX:

```c
void ACDXGeneratePrograms(
    ACDXFixedPipelineState* fpState,
    const ACDX_FIXED_PIPELINE_SETTINGS& fpSettings,
    ACDShaderProgram* &vertexProgram,
    ACDShaderProgram* &fragmentProgram)

void ACDXGenerateVertexProgram(    
    ACDXFixedPipelineState* fpState,
    const ACDX_FIXED_PIPELINE_SETTINGS& fpSettings,
    ACDShaderProgram* &vertexProgram)

void ACDXGenerateFragmentProgram(    
    ACDXFixedPipelineState* fpState,
    const ACDX_FIXED_PIPELINE_SETTINGS& fpSettings,
    ACDShaderProgram* &fragmentProgram)
```
The first function generates a vertex ‘vertexProgram’ and a fragment ‘fragmentProgram’ shader, written in Attila Byte Code, from a given T&L state provided by the two structures previously described, ‘fpState’ and ‘fpSettings’.

Apart from this tool, the ACDX also provides another one to translate a shader from the ARB shader language to the Attila Byte Code. The problem with openGL is that it uses two different constant tables, local and global. For this reason the compiling process is divided in two different steps.

First, the ARB shader code is compiled to the Attila Byte Code using the

\[
\text{ACDXCompiledProgram* ACDXCompileProgram(const std::string& code)}
\]

which generates an ACDXCompiledObject. This object, apart from containing the compiled ARB code, it also contains a new and unique constant table where each position is bind with another one from the old constant tables. This table doesn’t contain any value, as the only input of the function is the shader code, only a binding where to find the value.

Then the second step is fill the constant table with the appropriate values and this is done using the following function provided by the ACDX:

\[
\text{ACDXResolveProgram(const ACDXCompiledProgram* cProgram, const ACDXConstantBindingList* constantList, ACDShaderProgram* program)}
\]

The input parameters of this function are the ACDXCompiledProgram object ‘cProgram’ and new object called ACDXConstantBindingList ‘constantList’. There is also an output variable, ‘program’, which is an ACDShaderProgram object that contains the Attila Byte Code and the final constant table.

From this function it is necessary to explain how the ACDXConstantBindingList is built. This structure is nothing more than a list of ACDXConstantBinding objects. This object contains the information of one constant and is created as follows:

\[
\text{ACDXCreateConstantBinding (ACDX_BINDING_TARGET target, acd_uint constantIndex)}
\]

The ‘target’ is used to identify from which constant table the constant value comes, and the possible values are ACDX_BINDING_TARGET_LOCAL, if the constant is inside the local constant table, or ACDX_BINDING_TARGET_ENVIRONMENT, if the constant is from the enviroment constant table. The ‘constantIndex’ value is the identifier of the constant inside its constant table.
Then before issuing the ACDXResolveProgram method it is necessary to create for each constant value inside both constant tables an ACDXConstantBind and insert them inside the ACDXConstantBindList using the typical list operations.

Resource structure

The AOGL supports 3 different types of resources: textures, vertex buffers and shaders. The available texture types are 2D, 3D and Cube Maps.

The resources are also managed like a state machine. To work with one of them, the first thing to do is select it using the glBind* (GLenum target, GLuint id) function, where the asterisk can be Texture, Buffer and Program. This function is used to select which is the resource currently active. Each resource type has its own state, so there is the current Program, the current Texture and the current Buffer. The parameters of the functions are used to select the resource that is being set as current. The ‘target’ parameter is the resource subtype and in case of the Texture resource there are available the GL_TEXTURE_2D, GL_TEXTURE_3D and GL_TEXTURE_CUBE_MAP, in case of the Buffer the GL_BUFFER_ARRAY and GL_ELEMENT_ARRAY_BUFFER and in case of the Shader it can be GL_VERTEX_PROGRAM or GL_FRAGMENT_PROGRAM. Finally the ‘id’ parameter is used to select, from the resource subtype, which one it is being selected.

Then, every operation that involves any modification in any resource, it is done on the resource that is currently selected.

The AOGL needs some structures that emulate this behavior. The way how these structures are build is similar how it has been done in the ACD. There are some classes that contain the common characteristics and then, for each available resource, a new class inherits from it adding the particular characteristics of the resource. Then, there are available the ‘Base’ class from which each specific resource (‘Texture’, ‘Shader’ and ‘Buffer’) inherits.

The first element needed is somewhere to store the resource information. This element is called the AGLBaseObject. It is basically used to store the ACD object where the resource is saved. These objects belong to a given target, and this relation must be added. A new AGLBaseTarget object is introduced. This class contains all the available AGLBaseObjects that belong to the target. Finally it is necessary an element that stores all the available AGLBaseTargets and AGLBaseObjects and also saves which is the current AGLBaseObject set. This task is done by the AGLBaseManager.
The way how these structures are built using two different levels is very logical as the way how the resources are managed is very similar. So the Base objects implement the basic behavior common in all the available resources while each specific object implement its specific behavior.

This schema shows how this is done:

![Diagram showing the relationship between AGLBaseManager, AGLBaseTarget, and specific objects like AGLObject, AGLShaderManager, and AGLBufferManager.]

**Figure 48**

Next sections explain in more detail how each structure is built. First of all they explain how the Base structure is designed, and then continues explaining how each specific subclass is designed.

**AGL*Object**

The AGL*Object class contains all the information about the resources available in openGL. The question is what characteristics can be shared inside the AGLBaseObject, and which one are included inside each specific resource.

First of all, it is necessary to understand that each type of resource has its own structure to store the resource data, so this information cannot be saved inside the AGLBaseObject. Then this information must be saved inside each specific AGL*Object. The objects used to save the data are the ACDBuffer, the ACDShaderProgram and the ACDTexture objects. Remember that many of them contain much other information apart from the data of the resource.
Much information is included inside each specific resource object, so only a few things can be saved inside the AGLBaseObject. The most important one is the ID of the resource. This information is very important as it identifies the object among others inside the same target. Another information that is save here is the relation that the AGL*Object has with the AGL*Target. Apart from these parameters, this class also have getters and setter to obtain and modify this two values.

The rest of the parameters are declared in each specific AGL*Object. Each object contains the ACD data object used to store the resource data.

The AGLProgramObject also contains the local constant table and the ARB shader code. The reason why this information is not saved inside the ACDShaderProgram is because it is necessary to compile it before adding them inside the ACDShaderProgram. Another information that is saved here is the ACDXCompiledProgram object seen when the ACDX was introduced in the previous section. This object is set after the ARB shader program is compiled for the first time. It is very useful because it saves a lot of time. The fact that openGL uses two different constant tables and that there are the information that changes more, is the reason why it is necessary to have the ACDXCompiledProgram. Imagine every time one constant is modified having to recompile and redo the constant resolution to obtain the ACDShaderProgram. As constants are the shader attribute that change more the best solution to save time is save the ACDXCompiledProgram obtained from the compilation. Then, in case the constants are modified, it is only need to redo the constant resolution. For the same reason, it is also saved the last ACDShaderProgram object, and in case nothing is modified from the previous compilation and resolution it can be reused without doing anything else.

The AGLProgramObject also contains the local constant table. The methods provided are getters and setters to modify the shader code and the local constant table and also one operation to compile and resolve the ARB shader program. How this compilation takes place have been explained when the ACDX was introduced in the previous sections.

The AGLBufferObject only saves the ACDBuffer object. The methods available are getters and setters and also some methods to modify the ACDBuffer content.

The function void setContents(GLsizei size, const GLubyte* data, GLenum usage) and void setPartialContents (GLsizei offset, GLsizei size, const void* data) are used to modify the ACDBuffer content. The first one updates the whole buffer setting new content while the second one updates partially the buffer, starting from an 'offset'. The parameters of these methods are the ‘size’ of the input data, a pointer to the ‘data’ and a ‘usage’ value that specifies where the data have to be placed, in system or GPU memory.
Finally, the AGLTextureObject is the most complex one. Once the ACDTexture* objects have been explained, we have seen that many all the characteristics about the texture are saved inside this object. Otherwise, other characteristics regarding how the texture is read are saved inside the ACDSampler unit. OpenGL doesn’t do the same division between these characteristics and include some of them that the ACD place inside the AcDSampler, inside the Texture resource. Then the AGLTextureObject have to save these characteristics as they cannot be saved inside the ACDTexture object. These characteristics are the wrapping mode, the texture filters modes and the LOD levels. To set and modify these characteristics the AGLTextureObject provide getters and setters.

Another characteristic, that openGL has and the ACD not, is texture format transformation. OpenGL allows the user to add a texture mipmap defined using one format and tell the openGL API to save it using another different format. As the ACD doesn’t provide this capability the AGLTextureObject have to implement these transformations. These transformations are quite simple and usually suppose adding more bits to the format, like for example converting from GL_DEPTH_COMPONENT to GL_DEPTH_COMPONENT24. The most difficult conversion is when it is request to compress a texture, but this is supported by the ACD, as previously seen. The available functions here are to add and update the mipmap levels of the texture.

This two functions:

```c
void setContents( GLenum targetFace, GLint level, GLint ifmt, GLsizei width, GLsizei height, GLsizei depth, GLuint border, GLenum fmt, GLenum type, const GLubyte* data, GLint compressionSize)

void setPartialContents(GLenum targetFace, GLint level, GLint xoffset, GLint yoffset, GLint zoffset, GLsizei width, GLsizei height, GLsizei depth, GLenum fmt, GLenum type, const GLubyte* subData, GLint compressionSize)
```

are used to add or update a mipmap level. The input parameters are the face ‘targetFace’ (only in case it is a cube map texture) and the mipmap ‘level’ to identify which mipmap is being added or modified, the internal format ‘ifmt’ that must be used to save the texture internally, the ‘width’, ‘height’ and ‘depth’ of the texture, the size of the ‘border’, the format ‘fmt’ used for the input texture, the ‘type’ used to save the samples, the texture ‘data’ and finally the ‘size’ of the texture in bytes only in case the input format is a compressed one. Other parameters that appear in the other functions are the ‘xoffset’, ‘yoffset’ and ‘zoffset’ that point from which point the mipmap level have to be modified.

AGL*Target
AGLBaseTarget structure is the simplest one. It only contains the target ID and two pointers, one to the current AGLBaseObject set in this target and another to the default AGLBaseObject. The default AGLBaseObject is used when no current AGLBaseObject is set. Available operations are setters and getters to these parameters.

Apart from the previously described, the AGLTextureTarget and the AGLBufferTarget don’t provide any other functionality.

Inside the AGLShaderTarget it is also saved the environment constant table. Other operations available are those to modify the constant table.

**AGL*Manager**

The AGLBaseManager is the basic structure to control the resources. It contains all the available AGL*Objects and is the responsible for creating them as well as all the available AGL*Targets. The AGLBaseManager adds a new idea called ‘group’. The groups are one level above the targets, so every level has its own targets. The aim of the groups is to emulate some situations where there are various resources to configure and each one is independent from the other. A good example is the texture units; and in this situation each texture unit is one group. Which group is being modified is also a part of the state, so there is a variable that mark which group is currently active.

One of the most important functions in AGLBaseManager is `BaseObject& bindObject(GlEnum target, GLuint name)`. This function select as current the AGLBaseObject with ID ‘name’ and bind it to the AGLBaseTarget with name ‘target’ within the current target group. If the AGLBaseObject does not exist, it creates a new one. After this method, the AGLBaseTarget with name ‘target’ of the current group, is selected as the current AGLBaseTarget of the bound AGLBaseObject.

Another interesting functions is `void selectGroup(GLuint targetGroup)` which is used to change the current group, and also the `BaseObject* findObject(GlEnum target, GlEnum name)` which finds an AGLBaseObject with name ‘name’ inside a given target.

Each specific AGL*Manager is very simple and only implement its own constructor to add the desired targets. Here is where the AGLTargetManager creates as many groups as texture units and for each texture unit the desired targets. Also the AGLTargetManager bind each group with a new object called AGLTextureUnit, where the other texture units characteristics are saved.

**AGLTextureUnit**
This object contains all the information about how the texture unit is configured. There is one object for each texture unit available. Many information from here is map to the ACDSampler.

Apart from this information, this object also saves some parameters that are used in the fixed function state. This parameters are if the texture coordinate generation is active and which mode is enabled and also some parameters to configure the combining mode that OpenGL

This object contains all the information about how the texture unit has to be configured. Not all this information is map to the ACDSampler object.
The previous sections explained how the ACD and the AOGL are designed. As it can be seen, they are very big structures with many details to take into account. Debugging both structures is a hard task and could take months, because even the small error could generate strange behavior, typically a black screen, and it is really difficult to find out what causes this error. Moreover, there isn’t any tool, such as Pix for DirectX, which helps you debugging a trace, so the only way to know what the trace do is reading and interpreting it without any help.

Luckily a previous openGL library has been done on the simulator. This may seem irrelevant, but in fact it is the most important advantage this project has. Having another library that has to configure the underlying hardware the same way as the ACD is very useful as it is possible to compare how both libraries have configured the hardware and find what is causing the error.

Then, the way the ACD is debugged is adding ways to obtain information about the state and compare this information with the information obtained from the old openGL library. So, to obtain this information some functions have been added to the ACD and the old openGL library to obtain information, but also it has been modified the driver to provide information about the values of the registers. These methods have been added in different levels of the stack to make it easier to find the error.

Another important thing is which are the steps followed to test the ACD and the AOGL. The main problem is that they are built from the scratch, so it is not possible to program everything and then start debugging. It is necessary to go step by step, programming some basic openGL calls and the entire infrastructure needed to run them, test them, and when they run properly continue with new ones. Working this way makes it easier to find errors because only a small set of functions is being implemented each time. Working this way is not compatible with using real game traces, because they are very complex and implement a great amount of calls. The best way to debug the stack is using small traces, which test only a small set of functionalities.
Then, testing the stack is nothing more than start from the simplest small trace and get that each small trace runs. Luckily, these small traces have been provided by other members of the Attila group so they haven’t been implemented in this project. Once all these traces run perfectly on the stack, it means that the basic functionalities have been implemented, so next step is start debugging real games.

The old openGL library support up to 6 games: Doom 3, Quake 4, Prey, Chronicles of Riddick and Unreal Tournament 2004. Prey and Chronicles of Riddick are games that make an intensify use of the fixed function capabilities, for this reason it has been decided not to spend time in trying to support them because new games don’t use fixed function capabilities and also because this games are very old. Then, the ACD is focused on supporting the other 3. Moreover, it has been decided to add a new openGL game called Quake Wars. These games have been implemented starting from Doom3, Quake 4, Unreal Tournament 2004 and finally Quake Wars.

For the small traces and the real games, the strategy of comparing the state in both libraries has been used. But Quake Wars is not supported by the old openGL library, so this technique hasn’t been use in this case. Debugging this game has been much more difficult because it has been necessary to interpret what the game does without any help.

The following sections describe which techniques have been used to debug the ACD.

**Batches and frames**

The first way to compare both libraries is comparing the images that the simulator output for every batch and frame (this functionality hasn't been added to the Attila stack as it was already supported).

This is very useful as it is possible to compare batch by batch how each library builds the frame, and find from which batch the ACD output a different batch. This technique is even more important and useful to debug Quake Wars as this game is not supported by the old openGL library so the only way to check if everything is correct is compare the images obtained with the ACD with those obtained from the openGL player.

This is the first step done when a game is being debugged. From this test it is possible to know which is the first batch that fails, and also, depending on the error, it is possible to obtain a first idea of what is causing the error.

Here are some examples of the outputs:
Once it is found a batch that fails, next two steps is comparing that both API are setting the same way the hardware. Then it is necessary to check that the resources and the state set by each API is the same in this batch.

Resource

It is important that all the resources used in every batch are the same in both API. Then it is necessary to obtain and compare the shader programs, the vertex buffers and the textures used.

The first thing is from where this information can be obtained. The first idea is obtaining it from the Driver, but the problem is that it is not possible to distinguish there which kind of data it is being synchronized because the Driver received raw data without specifying the type. Another place could be when each API synchronizes the resources. From this place is easy to dump the information of each resource. Then, it have been modified each API adding a new function that dumps the information of each resource. Extracting the information from the vertex buffer or from the shaders is easy as they can be represented as text. Textures are a little more difficult as they are image. They can be output also as text, but the natural way to compare them is using images, so the best way to output the texture is using imaged. The format used to output the images is .ppm which is very easy to manage as every texel is represented as four values: red, green, blue and alpha.

Once the information is obtained, the last step is comparing the data obtained from both APIs.

Here are some examples of the outputs:

GPU Registers

Both libraries write the GPU registers. To be fully equivalent the values of the registers before issuing the same draw call in both libraries must be the same. If all the values are the same, and the resources synchronized are also the same, the batch obtained must be exactly the same, as the underlying hardware has been configured the same way.
The best way to track the GPU register values is modify the Driver adding to it a mechanism to track the values of the registers. This is the best way because only modifying the Driver the information can be obtained.

The writeGPURegister() Driver function doesn't store the value of the register being written, it only transform this writing into the appropriate AGP transaction. Then to be able to track the values of the GPU registers it is necessary to add a new data structure to the Driver. This structure is basically an array where the each GPU register is store with in his current value. Then, every time a writeGPURegister() is received, apart from issuing the appropriate AGP transaction it modifies the current value of the register being written. Finally, when a draw call is issued, this array is dump into a file. This method is the same in both APIs.

Then debugging a game is very simple. Just select a game and simulate it using both APIs. For every batch of the frame, a file with the all value of the GPU registers is output. Then debugging the ACD is as easy as comparing both outputs. When a register is different this is may be error. It is necessary to take into consideration that both libraries allocate the resources in a different way, so the MD used may be different, not being this an error.

This is the most powerful way to debug the games, as it is easy and quickly.

Here are some examples of the outputs:

XXXX
10. Economic planning

This chapter explains how much work and time the project has cost. The information provided here is not accurate but it helps to evaluate which tasks of the project have taken more time.

Three different roles have taken part in this project. The first one is the analyst, who has designed all the system, the second one is the programmer, who has programmed the specification provided by the analyst and finally the tester who have tested the whole stack using the tools seen in the previous chapter.

This chapter is divided in two different sections. The first one analyzes how the project has been scheduled in different tasks and how many hours each part took. The second part computes how much the project would have cost taking into consideration the schedule seen in the first section.

**Project scheduling**

This chart contains a list of the different task done in this project and how many our each one took.

<table>
<thead>
<tr>
<th>Work</th>
<th>Analyst</th>
<th>Programmer</th>
<th>Tester</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project definition</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Study of the Attila Simulator</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Study of the available API at Attila</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>openGL study</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DirectX 9 study</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACD design</td>
<td>150</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The total amount of time spent in this project is about 1360 hours

**Economic cost**

Next step is compute how much this project would have cost in case it wouldn’t be a PFC project. To compute the cost it is necessary to establish how much money each role earns.

- **Analyst**: 40 euros / hour
- **Programmer**: 30 euros / hour
- **Tester**: 25 euros / hour

Then, from the information previously obtained the cost of this project would have been the following:

- **Analyst**: 40 euros / hour * 620 hours = **24,800 euros**
- **Programmer**: 30 euros / hour * 550 hours = **16,500 euros**
- **Tester**: 25 euros / hour * 190 hours = **4,750 euros**

In total is **46,050 euros**.
11. Conclusions

As a result of this project a new API architecture has been designed to solve the problems found on the previous one. This new architecture is divided in two different levels; the lowest level is the Attila Common Driver (ACD) which main objectives are managing the state of the underlying hardware as well as the resources. The ACD provides an interface to the levels above it. As the ACD provides a non-standard interface, in order to play real workload, it is necessary to implement some units that translate from each specific API interface to the ACD one. These units are placed in another layer above the ACD. As the best way to debug the ACD is using real workload the openGL API is implemented on top of the ACD and is called Attila OpenGL (AOGL).

The main benefits obtained from the ACD are:

- Low level API operations such as resource and state management are only implemented once, inside the ACD, so it is only required to maintain and debug a unique version.
- New APIs don’t have to be implemented from scratch; basic resource and state management are provided by the ACD.
- API optimization are placed inside one place, the ACD, and not in every API implemented in the system
- New tools to debug the ACD and the above layers.
- Less work to main the entire architecture

As a result of implementing the AOGL, about 200 API calls are supported. The main capabilities supported with all these calls are:
- `glBegin()/glEnd()`, Vertex Arrays and Vertex Buffer Objects (VBO) for geometry attributes
- Available texture types: 2D, 3D and Cube Map
- Support for S3 Texture Compression (S3TC) from RGB and RGBA to DXT1
- Support for different texture types including all the S3TC types
- Texture filtering modes supported: Nearest, bilinear, trilinear and anisotropic
- Stencil test, depth test, alpha test and blending functions
- Basic fixed function support
- ARB vertex programs and ARB fragment programs

With all these capabilities, the AOGL+ACD support 4 different openGL games: Doom 3, Quake 4, Prey and Quake Wars.

Here you can found some snapshots obtained from the Attila simulator using the ACD.

Figure 49: Quake wars
Figure 50: Prey

Figure 51: Quake 4
When this project finished, other members of the Attila team start implementing DX9 on top of the ACD. In a very short time they achieved great results.

The main benefits obtained from using the ACD have been not having to implement the low level API operations as well as using a highly debugged layer that perform these tasks. Also, as they don’t need to perform low level management, they don’t need a deep knowledge of the underlying hardware.

Implementing DX9 prove that the requirements agreed at the beginning of this project have been met and justifies that the design used for the ACD is appropriate and works as expected being used by other people.

With a few months the D3D9 specific API supports for up to 12 new games. Here are some screenshots obtained from them:
12. Bibliography


Solis, C. (2007). Extensión a Direct3D del driver de un simulador de GPU. Barcelona: UPC.