Image complexity sensitive real-time rendering

Emil Jönsson  
dt06ej9@cs.lth.se  
Christofer Malmberg  
dt06cm9@cs.lth.se

June 11, 2013

Master’s thesis work carried out at  
the Department of Computer Science, Lund University.  

Supervisor: Michael Doggett, mike@cs.lth.se  
Examiner Flavius Gruian, Flavius.Gruian@cs.lth.se
Abstract

As display resolutions continue to increase, expensive computations made on every pixel start to become less necessary for the impact of the overall resulting image. This master thesis investigates a new way of reducing the computations of certain pixel areas of a rendered 3D image, either areas of low complexity or those of less importance to the output, using OpenGL. The resulting algorithm will contain a number of rendering passes, where the key pass will be constructed using the compute shader functionality, added in the OpenGL version 4.3.

By storing the geometry of a scene in multiple renderbuffers with different resolutions combined with a basic edge detection process, high- and low-complexity areas of a scene can be determined and rendered using the corresponding buffer. The resulting image will be composed of both high- and low-complexity areas, as well as the possibility of adding further stages to smoothen the transition between selected resolutions.

There are not any other works done in this field of rendering images. But since each new buffer created more OpenGL calls the performance was not improved.
Acknowledgements

Thanks to our supervisor Michael Doggett for his guidance and support throughout the making of this project.

Thanks to Magnus Andersson for creating, and providing assistance in the use of, the RenderChimp framework enabling us to use it as a starting point.

Thanks to fellow student Cem Eliyürekli for his support and advising during the initial part of the project.
Contents

1 Introduction ........................................ 7
   1.1 Resolution ........................................ 7
   1.2 Performance ....................................... 8
   1.3 Improving performance ......................... 10
   1.4 Division of work ................................ 10

2 RenderChimp ........................................ 11
   2.1 Scene Graph ...................................... 12

3 Sobel ................................................. 15
   3.1 Sobel filter ...................................... 15
   3.2 Edge and details ................................ 16
       3.2.1 Sobel filter Example 1: High resolution .. 17
       3.2.2 Sobel filter Example 2: Low to high .... 19
       3.2.3 Sobel filter Example 3: Light ............ 20
   3.3 Sobel Shader ..................................... 21
       3.3.1 Vertex Shader ................................ 21
       3.3.2 Fragment Shader ............................. 22

4 Compute Shader ...................................... 25
   4.1 OpenGL ........................................... 25
       4.1.1 Functions & Inputs ......................... 27
       4.1.2 Shared memory ............................. 28

5 Results .............................................. 29
   5.1 Deferred Shading ................................ 29
   5.2 Compute Shading ................................ 30
   5.3 GLReadPixel ..................................... 33
   5.4 Performance ..................................... 36
   5.5 Known Issues ................................... 36
Scene rendering is happening in many different fields. It could be everything from rendering pictures that people sent to each other, to rendering movies from a vacation with the family. In any case the rendering should be fast whilst still maintaining a good quality of the scene. In order to obtain a good quality you want to use a good resolution, but a good resolution means that the pixel count will be high. In turn, a high pixel count means that the amount of data the computer needs to process will also be high. Naturally, with a low resolution the pixel count will be smaller and the computer can process the data much faster.

An ideal solution then would be to have a competent mix of both. Have a high resolution where it is needed and a low resolution for the rest of the scene, in order to make the most of both quality and processing time. Then the resulting image would be composed of high resolution pixels on more detailed areas and the low resolution pixels on areas consisting of similar coloring, that is areas of low complexity.

1.1 Resolution

The resolution of a screen is the number of distinct pixels that can be displayed. It is usually measured by the number of pixels used in the width of the screen times the number used in the height. For computers there are some standard display resolutions that are commonly used for various applications, some of them listed below.

<table>
<thead>
<tr>
<th>Width</th>
<th>Height</th>
<th>Pixels</th>
</tr>
</thead>
<tbody>
<tr>
<td>640</td>
<td>480</td>
<td>307 200</td>
</tr>
<tr>
<td>1024</td>
<td>768</td>
<td>786 432</td>
</tr>
<tr>
<td>1280</td>
<td>720</td>
<td>921 600</td>
</tr>
<tr>
<td>1280</td>
<td>1024</td>
<td>1 310 720</td>
</tr>
<tr>
<td>1600</td>
<td>1200</td>
<td>1 920 000</td>
</tr>
</tbody>
</table>
1. Introduction

The number in height tells how many distinct pixels that will fit in the height of the screen. As can be seen with the lower resolution there are 640 pixels on the height of the screen and 480 distinct pixels at the width of the screen. At a higher resolution the number of pixels increase which means that a scene would need more calculations than when being rendered on a screen with lower resolution. Comparing the lowest with the highest resolution you get the following, \(\frac{640}{320} = 2.5\) and \(\frac{1280}{640} = 2.5\), whilst \(\frac{1920000}{307200}\) gives 6.25. The difference of height and width give a low value but the total value of pixel, low against high resolution, is high. Figure 1.1 shows two different kind of resolutions and the different artifacts that lower resolutions give.

![Figure 1.1: Comparison between high and low resolution outputs. Top is 720x480 and bottom is 1680x1050.](image)

1.2 Performance

The aim with the computers of today is to make them faster so they can solve problems faster, but the speed still depends on the quality of the work. If the
computer is solving a mathematical problem the speed will depend on how many
data-points the computer has to work with. If the data consists of just a few data-
points then the quality of the mathematical problem often would not be that precise.
On the other hand, if the computer uses more data-points then it will give a higher
quality solution, but the time to compute it will increase as well.

The same is true when working with computer-rendered scenes. As mentioned
above, when the resolution is high the computer has more pixels to calculate and
work with, compared to a lower resolution. The Frames Per Second (FPS) will be
lower the more pixels the computer has to work with. FPS is a way to calculate
how fast a computer can render a scene, depending on the graphical settings. When
measuring FPS in games the count should be high, meaning that the computer can
render many scenes per second so the person looking at the screen does not come
across any graphical 'jumps' or glitches when moving around.

![Image of FPS comparison](image)

In figure 1.2 the right side has a higher resolution and thus results in a lower
FPS compared to the left side which has a lower resolution but a higher FPS.
1.3 Improving performance

When rendering scenes you want the computer to give a good graphical view of the work while still being able to give a good performance. This thesis is about combining these two qualities, that is have a good visual output of the scene whilst still have the computer render at a reasonable speed. This is done by combining different resolutions into one scene and then render that scene. Start with a low resolution scene at first, followed by calculating what is important in the scene and render those areas at a higher resolution. For example, if the scene to be rendered is a scene of a field and a tree, then the tree would most likely be the complex part of the scene and maybe not the sky or the green grass. What the program does then is to scan for areas where there are details that should be rendered in a higher resolution, in this case the tree, which will be given a higher resolution. This way of rendering will theoretically give a higher FPS output since the scene as a whole is being rendered at a lower resolution and only the important parts of the scene will be rendered in full or a higher resolution.

1.4 Division of work

In the beginning there was a lot of reading and research to be done and this was therefore carried out together. However, as the coding began we started to focus more on different parts and therefore divided the work to be done. It was decided that Christofer was going to focus on the Sobel algorithm, and an initial approach to the problem without using the compute shader functionality (See chapter 4). This includes the method glReadPixels, an experimental merge shader for mixing resolutions and the setup of the framework. Additionally, have a look at a light and shadow shader to be tested with the Sobel algorithm. In the meantime, Emil would focus on getting the new compute shader functionality to work with the chosen framework, and later on integrating it with our program. As well as digging into RenderChimp and understanding how it was working internally, in order to make it accept the compute shader.
Chapter 2
RenderChimp

RenderChimp is a graphics programming framework, designed to enable construction of cross platform graphics applications for educational purposes. It was created by Magnus Andersson with the help of some members of the Lund University Graphics Group, to work as an environment for assignments given in certain graphical programming courses at Lunds Tekniska Högskola. In the beginning it was mostly based on the concept of the sceneGraph object, which was basically a way to encapsulate calls to OpenGLEs in order to enable easy graphics programming for mobile units. It has since then evolved towards more of a general graphics engine, enabling the support of shaders and recent OpenGL technology [1]. Figure 2.1 shows a diagram of the system setup of RenderChimp, most importantly how the applications does not have any direct contact with the low-level graphics.

```
/--> [ RenderChimp ]
  \\
[ Application ]    v
\--> [ RC::Renderer ] --> [ Low-level graphics ]
                 (OpenGL)
                 (OpenGL ES)
                 (Software ray tracer)
                 (...)
```

Figure 2.1: A high-level diagram of the RenderChimp system, image courtesy of Magnus Andersson [3].

When using RenderChimp it is worth noting that the application has no direct contact with the low-level graphics API. Instead, all rendering calls and state changes flow through the scene graph setup and eventually end up in the Renderer, which
2. RenderChimp

handles and relays all rendering related matters. Due to this design it is worth mentioning that the application should never make direct calls to the low-level API, as this may result in weird behavior and/or crashes. The most commonly used renderers at this time are the OpenGL ones, but support for other renderers has been added such as a CUDA ray tracer and a software rasterizer. Users are welcome to add new renderers as long as they comply with the header file, although it is not assured that they will work flawlessly \[3\]. There are three functions that need to be included in each RenderChimp program application. They are as follows:

- **RCInit()** - the setup phase which is called when the program is loaded. Various objects are defined here, such as scene graph objects, shaders, textures, etc.

- **RCUpdate()** - called every frame when the program is running, and as the name suggests, updates the frames with information from the geometry and shaders.

- **RCDestroy()** - called once when the program is terminated and deletes the scenegraph objects recursively.

### 2.1 Scene Graph

The scene graph concept is a main feature in the RenderChimp framework. It can be referred to as the "almighty singleton object" as all modifications to nodes and resources must go through this object. Nodes and resources then, are the building blocks in the scene graph setup, and are what the user comes in contact with when constructing a program. Nodes are the foundations, describing the hierarchical structure of a scene. One node represent one object and they are relatively cheap while rendering. Whereas a resource describes certain data, it can be organized in many ways and can also be instantiated more than one time. They can however be relatively expensive while rendering, depending on the resource used. The goal of these two is for them to be used in coherence with one another to reduce data. For example, you want to create a game with multiple identical enemies. Instead of storing the same data multiple times – share it. The resulting effect is that the same data will be located at different positions in the scene \[2\ p. 29-31\]. Figures \[2.2\] and \[2.3\] show example nodes and resources that are commonly used in a standard application constructed in RenderChimp.
2.1 Scene Graph

Figure 2.2: A few of the available scene graph nodes and their hierarchical setup, image courtesy of Magnus Andersson [2, p. 27].

Figure 2.3: An example line-up of available scene graph resources, image courtesy of Magnus Andersson [2, p. 28].
Chapter 3
Sobel

The Sobel operator is an edge detection algorithm using the idea of convolution matrices [12] for an image. The algorithm will give a gradient for each pixel from the convolution matrix and that pixel will take a new value depending on the surrounding pixels. The Sobel operator will result in a buffer and with that buffer has the detailed sections highlighted so the computer knows what is complexed. These sections will then be used to check against a threshold to calculate which resolution they will have.

3.1 Sobel filter

In order to detect edges in the real-time rendering scene we have decided to use a Sobel filter [5]. The Sobel filter is one of many different algorithm which is used to detect edges in images. Since Sobel has a good result of highlighting the difference of pixel by having high values in the convolution matrices we choose Sobel. The filter would make use of the Sobel operator, which is a way of highlighting edges in a scene as it gives a pixel new content values depending on the surrounding pixels [7]. The way it works is that when it looks at a pixel, it looks at how the pixel differentiate depending on the surrounding pixels. The operator consists of two matrices, one which looks at the horizontal derivative of the pixel value and one which looks at the vertical derivative. The matrices are as follows

\[
g_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad g_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}
\] (3.1)

These matrices are then multiplied with the original image, A, giving two new matrices.

\[
G_x = g_x \ast A
\] (3.2)
3. Sobel

\[ G_y = g_y \ast A \]  
(3.3)

Then those two matrices are added together

\[ G = \sqrt{G_x^2 + G_y^2} \]  
(3.4)

Resulting in the gradient matrices of that point. Then every value is added together and put in the place of the pixel that was observed.

![Sobel convolution matrix](image)

Figure 3.1: Sobel convolution matrix

Figure 3.1 shows how the Sobel matrix works over an image. At the left most part is the image with pixel values. The 3x3 matrix is the Sobel operator convolution matrix. This example only has one Sobel operation matrix instead of two. After the multiplication is done it is time to sum them up. The equation is 

\[(40 \ast 0) + (42 \ast 1) + (46 \ast 0) + (46 \ast 0) + (50 \ast 0) + (55 \ast 0) + (52 \ast 0) + (56 \ast 0) + (58 \ast 0) = 42.\]

The new value of that pixel then is 42. [6][12]

3.2 Edge and details

When rendering games on computers of today, the game usually renders at a fixed resolution. This is because it is easier to set a resolution and change the buffer size to a specific resolution when rendering scenes. The resolution can be everything from 640x480 all the way up to 2560x1600 (and maybe even higher than that). With a lower resolution the final image will be blurry or "pixely", that is you are able to see the edges of the pixels. The computer will render these scenes at a faster rate since the number of pixels is lower. With a higher resolution, the final image will include more details, and the visual output will be better. However, the problem with too high resolution is that the pixel count will be higher as well, and the computer may have a hard time rendering the scene.

Therefore, to combine the two advantages into a single graphics program would be optimal. Have a low resolution for the rendering of the scene as a whole, which means that it will be fast, whilst still having a higher kind of resolution for the more detailed areas in the scene.

The Sobel filter algorithm [5] will give the computer a way to detect where the details of an image are located. First, the scene is rendered in lower resolution to give an outline of how the image will look. Then use the Sobel filter on the lower resolution to calculate where the edges are, and encapsulate the complex areas depending on the algorithm. When the Sobel filter has run its course the computer will know where the complex areas are and render certain areas to higher resolution.
3.2 Edge and details

3.2.1 Sobel filter Example 1: High resolution

The first example of how the Sobel filter works will be a high resolution scene of sponza, figure 3.2. The following image series shows a scene rendering at a resolution of 1024x768. There are sections in the image which are complex and sections where there is no need for a high resolution.

![Figure 3.2: Sponza scene, rendered at full resolution.](image)

After the scene has been rendered, the computer will give the scene over to the Sobel filter. The standard Sobel filter works on an edge detection level. If there is a change in pixel color it will get highlighted by the Sobel filter. In figure 3.3 the white pixel are highlighted pixels.

![Figure 3.3: Same scene with a Sobel filter applied to it.](image)

The Sobel filter gives a scene, figure 3.3 where a change in color results in a highlighted white pixel. This type of filter does not highlight where the details are since almost the whole scene is highlighted. This means that we need to modify our Sobel
algorithm slightly. One way is to look at the depth values of the pixels which is render with the texture values in geometry buffer. It would work by looking at the values and then comparing them to that of the surrounding pixels. If, for example, the pixel currently being rendering is located on the ground in the scene the surrounding pixels which are also on the ground will have the same depth value and we will not have a highlighted ground. This means that the differential between each pixel will be small. Added together they will not surpass the threshold and therefore not become highlighted.

![Figure 3.4: Same scene rendered with a Sobel filter checking the depth values. The bottom image also contains checking of the normal values.](image)

The Sobel filter algorithm that checks the depth values instead of the edges is the
better choice. As can be seen in figure 3.4 compared to figure 3.3 there are fewer highlighted pixels, which makes it easier to make out the edges in the scene. One way to make the Sobel filter better at detect complexity in the image would be to add the normals to the depth check as well. If there is a change in the normal value, then the cross product of those normals will give a highlight pixel. The resulting Sobel filter algorithm with normals does not just highlight the same areas as the previous filter, but have a few more areas where the normal values have a larger impact. For example, there are more highlighted pixels at the arcs since the normals at the arcs will point in different directions.

3.2.2 Sobel filter Example 2: Low to high

In the previous section the computer rendered everything at a full screen-resolution level. Since it would be more interesting to render the scene at a lower resolution, and then have a higher resolution at the detailed areas, this example will take a look at the transition from a lower to a higher resolution. This example will show the process of first having a lower resolution scene, figure 3.5, then apply the Sobel filter algorithm and give the complex areas a higher resolution.

![Figure 3.5](image.png)

Figure 3.5: The Sponza scene rendered at low resolution applied to a fullscreen output.

This is the same scene as before but rendered at an eight of the full resolution. The areas which would be interesting, like the lion at the end of the scene or the column, are blurry. The Sobel filter produces the following scene, figure 3.6. The Sobel filter has highlighted the details in the low resolution scene. The program then changes these highlighted pixels into the full screen resolution. Figure 3.7 shows a scene with two different kinds of resolution. One blurry where there are not many details and the fullscreen resolution at the columns where there are details.
3. Sobel

Figure 3.6: Same scene rendered at low resolution with an applied Sobel filter.

Figure 3.7: Rendered scene at full resolution, a mix between a low resolution base and high resolution areas, as determined by the Sobel filter.

3.2.3 Sobel filter Example 3: Light

The last example will be looking at scenes with lights. Some scenes may have different kinds of light that will increase the brightness in certain parts of the scenes, or may have spotlights that will shine on a wall, highlighting that wall. If a pixel is highlighted by the light source, it is not interesting to look at the surrounding pixels in the light buffer, but instead see if the pixel has light or not. Because of
this, the value from the light buffer will be checked against the threshold and not get a gradient matrix equation \[ 3.4 \]. The light value will be added to the gradient matrix that has the normal and depth values. Because the light values in the buffer are always quite high, a pixel that has light on it will pass the threshold value. In a room which is dark a lighted area will stand out and must be in higher resolution.

![Image of Sponza scene rendered with Sobel filter](image)

Figure 3.8: The Sponza scene rendered at full resolution with spotlights with an applied Sobel filter. Each circle is a spotlight highlighting an area.

Figure 3.8 shows the sponza scene with a depth buffer, a normal buffer and a light buffer used together with a Sobel filter. The new circles in figure 3.8, are spotlights and highlight certain areas.

### 3.3 Sobel Shader

Since RenderChimp will highlight each pixel that has details in them, the program must do this on a shader level. Each pixel must go through the Sobel filter and be assigned the kind of resolution the pixel will get.

### 3.3.1 Vertex Shader

The vertex shader will not have that much work to do. Since the Sobel filter will work on a pixel level the vertex shader just points out where in the vertex the fragment shader will calculate the pixel.

```cpp
attribute vec2 Vertex;

varying vec2 tc;

void main()
```

21
3. Sobel

```cpp
{ 
    tc = 0.5 + Vertex * 0.5;
    gl_Position = vec4(Vertex, 0.0, 1.0);
}
```

### 3.3.2 Fragment Shader

It is in the fragment shader that the computer will do all the work by calculating the details of the scene. From the vertex shader the fragment shader knows which pixel being calculated at the time. This next part will explain what the fragment shader does. The code is located in appendix B.

To get a Sobel filter to work, the computer needs to look at the surrounding pixels, figure 3.1. At the start of the fragment shader the computer generates a grid for the surrounding pixels, with the observing pixel situated in the middle. This will be used for the convolution matrix. With this the computer knows which pixel is being observed and the position of the surrounding pixels. These pixels are then saved in a vector and used to look up position from the buffers.

```cpp
for (int i = -1; i <= 1; i++)
    for (int j = -1; j <= 1; j++)
        create grid around surrounding pixel
```

The next part in the fragment shader is to extract the grid of pixel data from different buffers. The buffers in use here are the depth buffer and the normal buffer. The shader also use the light buffer for the pixel that is being observed. Since the pixel is only interesting if the pixel is in light, the shader does not use the grid on the light buffer. The depth buffer is encoded in three different values and a method called "EXTRACT DEPTH" is used to get the single depth value for the matrix. The normal is normalized to be between values $x \in [-1, 1]$. To know the difference in normals the cross product [14] is used to detect whether the normals 'point' in the same direction. After this step the shader has the two matrices that will be multiplied with the Sobel matrices as in equations 3.2 and 3.3.

```cpp
for (int i = 0; i < 9; i++){
    extract value from depth buffer
    extract value from normal buffer
    decode depth value
    normalize normal value
    cross product on normal value
}
```

The shader then creates the two convolution matrices, equation 3.2 and 3.3. These convolution matrices are multiplied with the matrices that hold the depth values and normal values. Which means that after this step the shader have four matrices, two matrices for horizontal edges, normal and depth, and two matrices for vertical edges. These matrices are then added together with equation 3.4. Before being compared to the threshold the light value of the observed pixel is added. The final value is then being compared to the threshold for the Fragment shader.

```cpp
create Sobel convolution matrices
```
multiply Sobel convolution matrices with image

... for (int i = 0; i < 3; i++)
  for (int j = 0; j < 3; j++) {
    add values from different matrices together
  }
sum = 0.1 * sqrt (sumxN * sumxN + sumyN * sumyN);
sum += 0.1 * sqrt (sumxD * sumxD + sumyD * sumyD);
Chapter 4
Compute Shader

Over the last few years there has been a large increase in the power of graphics hardware. Naturally, a desire to control and utilize this power for work that differs from the traditional graphics pipeline, has emerged. Lately, many different platforms have been introduced to try and satisfy this in order to make the most of it. A few well-known examples are:

- CUDA\(^{[11]}\) - a parallel computing platform created by NVIDIA, which makes their GPUs accessible for computations like CPUs.
- OpenCL - a kind of parallel computing language much like OpenGL is for graphics, which can be executed across numerous types of processing units.
- DirectCompute \(^{[9]}\) - an API created by Microsoft which supports general-purpose computing on GPUs and shares a range of interfaces with the previous two.

Although this kind of parallel computing has also been recently introduced as a type of shader. This makes it easier to use in coherence with the graphics pipeline without having to interoperate with another type of compute API (such as those listed above). This type was named Compute Shader.

4.1 OpenGL

In August of 2012 a new version of OpenGL was released, the main feature being the introduction of the Compute Shaders concept. In most aspects Compute Shaders are identical to all other types of OpenGL shaders, with similar status and other properties, but at the same time it is very different from standard shaders such as the vertex and fragment shaders. Other shader stages have well-defined input and output values, whether they are user-defined or built-in, along with an intuitive...
execution order based on the nature of that stage. In essence, vertex shaders execute once per vertex, fragment shader execution is based on the fragments created during rasterization, and so on [10].

The compute shader handles things differently. It does not have any user-defined inputs or outputs, nor does it have a fixed execution frequency. It is up to each compute shader to define when and how many times it will execute, based on the function used to execute the operation. This means that the space on which compute shaders operate is abstract, and the built-in inputs define where in the space of execution a particular invocation (thread) should be executed. These built-in inputs consist of the concept of a work group, which is basically how the user controls the amount of compute operations to be executed. So if a compute shader needs other types of inputs it is up to the shader itself to fetch that data, via textures/image load or shader storage blocks. Also, if it wants to actually compute something it must explicit write to an image or a shader storage block [10]. Figure 4.1 below shows the updated outline of the standard OpenGL pipeline, with added support for how compute shaders interact with the rest of the components.

The work groups operate on a 3D space which means that they have a number of "X", "Y" and "Z" groups. If three dimensions are not necessary for the operation in question, the user can simply assign the redundant group(s) the value 1, enabling one- or two-dimensional computations instead. Within each work group there may be several compute shader invocations, the amount defined by the shader itself. This is called the local size of the work group. Much like the work group, the local size can be one-, two- or three-dimensional depending on what is needed for the operation [10].

For example, if a compute shader has a local size of (128,1,1) and a work group count (dispatch) of (16,8,64), then there will be 1,048,576 separate shader invocations, each having a unique set of inputs [10]. The reason for making a distinction
between these two sets is that different invocations within a work group can inter-
communicate via the use of shared variables. While communication between work
groups is possible, it requires global memory synchronization. Figure 4.2 shows the
relationships between a single invocation, the layout of a work group and how they
are represented in the dispatch operation.

4.1.1 Functions & Inputs

As compute shaders are outside the traditional rendering pipeline, they do not work
with usual rendering functions. There is mainly one function to use in order to initi-
ate compute operations, and will select whichever compute shader currently active.
(There is a variation for stored work group information inside a buffer object, but
it is fairly similar).

```c
void glDispatchCompute(GLuint num_groups_x, GLuint num_groups_y,
                        GLuint num_groups_z);
```

The following are the built-in inputs that are available within a compute shader:

- `gl_NumWorkGroups` - number of work groups passed to the dispatch function.
- `gl_WorkGroupID` - current work group for this invocation.
- `gl_LocalInvocationID` - current invocation within the work group.
- `gl_GlobalInvocationID` - uniquely identifies a particular invocation among all
  invocations of the dispatch call.
- `gl_LocalInvocationIndex` - 1D representation of `gl_LocalInvocationID`, iden-
  tifies the invocation within the work group.

Figure 4.3 shows how a simple code example is interpreted by the invocations in
a work group.
4.1.2 Shared memory

Compute shaders provide the functionality of shared memory, through the use of shared variables. These must however be global, uninitialized and can not be just any arbitrary type, though common types such as int, float, arrays and structs are fine. Much like other GPU based parallel computing platforms, the shared variables use the rules for incoherent memory access. Basically, this means that there needs to be some kind of synchronization step to ensure that the shared variables are visible.

By using the function `memoryBarrierShared()` the order of shared variables will be controlled. However, there might still be a need to ensure that variables modified by invocations within a work group will be correct and therefore a way to synchronize the execution with the invocations. This is accomplished using the `barrier()` function, which enforces an explicit synchronization between all the invocations in a work group. Figure 4.4 shows a brief code example illustrating the need for shared memory barriers.

![Figure 4.3: Per-thread variable usage, image courtesy of Mark Kilgard, NVIDIA [8] p. 31.](image)

![Figure 4.4: Per-Work group shared variable usage, image courtesy of Mark Kilgard, NVIDIA [8] p. 33.](image)
Chapter 5
Results

The main goal of this thesis project was to measure the performance gain acquired when substituting low-complexity pixel areas with a smaller resolution output. A lot of methods to approach this hypothesis were looked into, as well as the question of what platform and coding environment were to be used. It was decided that we were going to use the RenderChimp framework, with an initial setup gotten from an assignment in one of the current courses in computer graphics. A main part of the project was to use the recently released version of OpenGL, that is the new compute shaders feature. The program would consist of a number of passes for easier comprehensibility. The scene would be a variation of the famous sponza scene, consisting of a lot of geometric objects it would be suitable due to its variety of complex areas. Using a simple edge detection filter based on the Sobel concept, certain areas defined by a tiling architecture would be considered high complexity and therefore be rendered with full resolution.

Most of the work would be carried out by the compute shader, it being the key stage in the process. The previous stage would only create the scene objects and the last stage would only render the output from the compute shader onto the screen. Then, using Renderchimp’s built-in FPS counter, measure a rough estimation of the performance gained through the output modifications.

5.1 Deferred Shading

The resulting RenderChimp program structure is based on the principles of deferred shading. The idea is to have a number of shader stages or passes, where no shading is performed during the first pass. The main advantage of using deferred shading is to reduce the computations necessary for lighting interactions with the geometry, that is light is only calculated for the parts of geometry where it actually hits. We decided to keep this structure even though we do not really utilize this advantage.
5. Results

Figure 5.1: The sponza scene with basic deferred shading.

The main part of the program can be broken down into three passes. Figure 5.1 shows the standard output of the sponza scene rendered using the principles of deferred shading.

**Geometry Pass:** The first pass in the program pipeline. The main objective of this pass is to create and setup the objects featured in the scene. Strictly speaking it only renders the geometry information of the scene constructed in the RCInit() method, to a number of defined geometry buffers. The buffers store various information about each pixel in view. Three important aspects of this information are vital for the next pass, mainly the depth & normal values along with the diffuse color of each pixel in view.

**Compute Pass:** The second pass which does most of the computational work. Using the different pixel values mentioned in the previous pass, it creates a basic edge detection filter based on the Sobel operator. Then, using a compute shader-based tiling architecture system, different regions of the output view of the scene are rendered either in full resolution or in a preset, smaller, resolution. This means that a region can consist of a lot of high-resolution pixels or one (or a few) low-resolution pixel(s). Each pixel is given a color based on its complexity and then written onto a image texture, that is passed along to the next pass.

**Copy Pass:** The third and final pass of the process. This is basically a render to screen pass, where the output from the compute pass is rendered onto the display. Due to the nature of the compute shader being mostly about computation and not so much about rendering, adding an extra pass for standard shader rendering seemed to be the logical choice.

5.2 Compute Shading

The second pass is the stage in the rendering process which does most of the work in the program. By continuously grabbing incoming values from the geometry buffer, based on the viewpoint it calculates the complexity of every pixel. Based on its resolution it then selects the appropriate color value from the available resolution buffers in use. In this section this process will be broken down in more detail. Every
pixel in the current viewpoint contains specific values needed to compute the edge detection filter, which decides if the pixel should be high or low resolution.

In order to make the most of the compute shader’s parallel computing abilities, the current viewpoint, in the form of an image, is broken down into tiles with a preset size. As mentioned in the previous chapter, by using the "shared" variable qualifier, a defined set of invocations will be in charge of computing the pixel values for a specific tile. By dividing the work in this manner, the illusion of parallelism occurs, due to the great improvement in speed.

To construct the edge detection filter based on shared memory, we have used the principles of image convolution. In order to sample the output color of pixels each output pixel must have access to neighboring pixels within a certain radius. This means that the tiles in the shared memory must be expanded with an apron that contains neighboring pixels. Naturally, since the edge pixels of the image will have neighbors representing data outside of the image, only pixels inside the apron will be written to the resulting image. Figure 5.2 depicts the relationship between the image, the tiling of the image and the aprons. In essence, it describes the usage of the apron when performing image convolution.

The various pixel information is then stored in shared arrays. These arrays are then multiplied with the Sobel matrices together with a preset weight value. The weight value is used to amplify or diminish the effects of the depth and normal values of the resulting edge filter. In our project we have used the value 50 for depth and 0.01 for the normals. Finally the stored information inside the aprons is looped and multiplied by the Sobel matrices which is then summed together. This sum is then compared with a threshold value (typically in the region of 0.1 - 0.9), which in turn decides which resolution buffer the pixel will get its color from. The color
value is then stored in the image using the imageStore() function. Example code of a compute shader can be viewed in appendix A.

When running the compute shader pass there are a few settings that define how the pixel information is handled. Basically, we will view these as either high or low mode depending on their impact on the resolution, with the possibility for more stages in between.

First, there is the dispatch function which regulates the number of invocations that the shader shall create. When in high mode, it spawns one invocation per high resolution pixel, making the application sluggish and unnecessary slow. While in the low mode, it takes into account the size of the tiles the image has been broken into, and spawns one work group per tile, consisting of a number of invocations. As the low mode enables easier control for the parallel computing concept it is the most suited for our application.

Second, we have the creation of the Sobel edge detection filter. In high mode it takes its component values (normals, depth and diffuse color) from the high resolution buffers, making it the better option precision wise at the cost of performance. The effects of the low mode are then easily deducible, that is the same but with low resolution buffers, better performance but with certain visible artifacts in the transition between pixel areas of different resolution. Figures 5.3 and 5.4 show a comparison between a specific part of the sponza scene, where certain parts should be rendered in low resolution and others in high resolution. The distinction between the two figures being what resolution buffers being used to calculate the Sobel filter in the compute shader.

![Figure 5.3: Comparison between standard and mixed resolution outputs with low resolution Sobel calculations. Mixed has an extra pass in the rendering pipeline, hence the reduced FPS. A closer inspection of the right image reveals the high resolution artifacts, for example the top-right area of the lion’s head and the transition between the ground and the wall in the bottom-left.](image-url)
5.3 GLReadPixel

We did not want to create a full resolution buffer for the whole scene. To split the scene in different tiles [4] and generate the picture for each tile would solve this. The scene was split into 4x4 tiles and for each tile we rendered that part of the scene. If that tile would pass the threshold it would be rendered in full resolution.

The problem with this solution is that each pixel value from GPU would need to be shared to the CPU and memory sharing between these two components is not that fast. GLReadPixel works in different ways depending on what the coder wants. A way to use it is to say that it will look at the buffer which is writing to the screen and add those pixels to a different buffer. There are also ways to give different buffers to the GLReadPixel [13] in a way which will make it read the whole size of the buffer. Below shows the code to extract pixel values from a buffer with GLReadPixel.

```c
1 glReadBuffer (GL_FRONT) ;
2 glCopyTexImage2D (GL_TEXTURE_2D, 0, GL_RED, 0, 0, worldX, worldY, 0) ;
3 glGetTexImage(GL_TEXTURE_2D, 0, GL_RED, GL_FLOAT, pixels) ;
4
5 glReadPixels(0, 0, worldX, worldY, GL_RED, GL_FLOAT, pixels) ;
6 for (i32 m = 0; m < 4; m++) {
7     for (i32 k = 0; k < 4; k++) {
8         sum[4*m+k] = 0.0 ;
9         for (i32 y = m*(i32)worldY/4; y < (m+1)*(i32)worldY/4; y++)
10             for (i32 x = k*(i32)worldX/4; x < (k+1)*(i32)worldX/4; x++)
11                 sum[4*m+k] += pixels[(y * worldX + x)] ;
12     }
9 }
```

The FPS when using this buffer size for GLReadPixel was down to four. Even if the size of the buffer that was used to render to screen was smaller, with the size of 4x4, it would not make the process any faster since the bottleneck is the sharing of...
pixels between the CPU and the GPU. After the pixel would have been read from
the GPU we would have an array of size 16, which contains the number of how many
values in that tile are highlighted to be in a higher resolution. The next step would
be to loop through the tiles and generate the higher resolution for the tiles with a
value higher than the threshold. Figure ?? have the code how rendering different
tiles in different resolution with glReadPixel.

```cpp
for (int i = 0; i < 16; i++) {
    setSpecificTiles(tileArray, tileBuff, u32(i));
    vertex_array = VertexArray(tileArray, 6);
    fullScreenQuad = VertexArray(vertex_array, true);
    if (summers[i] > 200.0) {
        Geometry::setRenderPass("GeometryPass");
        Renderer::setRenderTarget(geometryBuffer);
        Node *c = fullScreenQuad->getChild(i);
        fullScreenQuad->detachChild(c);
        world->attachChild(c);
        world->render();
        world->detachChild(c);
        fullScreenQuad->attachChild(c);
        world->attachChild(fullScreenQuad->getChild());
    }
    Geometry::setRenderPass("Default");
    Renderer::setRenderTarget(mergeBuffer);
    MergeBufferShader->setScalar("summer", summers[i], UNIFORM_FLOAT32);
    fullScreenQuad->setShaderProgram(MergeBufferShader, "Default");
    fullScreenQuad->setTexture("sobelBuffer", sobelBuffer->
        getTexture(0));
    fullScreenQuad->setTexture("geometryBuffer",geometryBuffer->
        getTexture(2));
    fullScreenQuad->setTexture("geometry2Buffer",
        geometry2Buffer->getTexture(2));
    fullScreenQuad->render();
}
setSpecificTiles(tileArray, tileBuff);
vertex_array = fullScreenQuad->getVertexArray();
vertex_array = VertexArray(tileArray, 6*16);
fullScreenQuad = VertexArray(vertex_array, true);
```

At the start of the for-loop a specific tile is chosen for this pass and created. If the
sum of the glReadPixel value is above the threshold then that tile will be rendered
in a higher resolution. That tile is then rendered with the full resolution buffer
and later attached onto the world object, which then will be written to the screen.
Depending on the version of the code we had a merge shader which took three
different kinds of buffers, rendered and used on the tile at hand. If the tile did not
have a full resolution buffer, it would be discarded, and after the loop is done every
tile is put together, resulting in a mix of full resolution tiles and lower resolution
The rendering of the full resolution buffer reduced the FPS of the program. Figure 5.5 shows the scene where we use the lower resolution Sobel filter but while still having a full resolution buffer rendered in the background. Figure 5.6 shows the same scene without the full resolution buffer. It becomes clear that the full screen resolution buffer affects the FPS of the program.
5.4 Performance

We have tested our application on the sponza scene with two different graphics cards, the GeForce GTX 480 and the GeForce GTS 450. On both cards, using some different scenarios involving the settings from the previous section, we observed the effects of the output both visually and performance wise. We observed both the program with only normal shaders, vertex and fragment, and program with the compute shader. The following table shows the difference between the two programs:

<table>
<thead>
<tr>
<th>Program</th>
<th>Type</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute shader</td>
<td>Full resolution</td>
<td>108</td>
</tr>
<tr>
<td>Compute shader</td>
<td>Without full resolution</td>
<td>160</td>
</tr>
<tr>
<td>Normal shader</td>
<td>Full resolution</td>
<td>105</td>
</tr>
<tr>
<td>Normal shader</td>
<td>Without full resolution</td>
<td>195</td>
</tr>
</tbody>
</table>

As can be seen by the table there is not a great improvement between using compute shader and normal shader. The frame count when the full resolution buffer was deactivated is higher on the normal shader but this is due to that shader generated a Sobel scene and the compute shader generated a texture scene. It is likely that the number of OpenGL call on RenderChimp lower the performance. Visually, the resulting output with mixed resolution areas works better with the high resolution Sobel filter.

5.5 Known Issues

There are a lot of factors in this project that have had an effect on the resulting performance and quality, most notably, the usage of RenderChimp. While it provides clear advantages in the handling and layout of the code, RenderChimp being what it is also results in a number of unnecessary calls in the setup stage. Tasks such as creating buffers or shader programs or simply calling a rendering function must travel through a pipeline of scene graph nodes or resources, to often end up in the same rendering class. In addition to this, RenderChimp still being in its evolution stage has a certain way of handling the rendering process' and is not always up to date with the latest versions, simply because it need not be due to its initial purpose.

Another issue was when trying to combine multiple buffers with different resolutions into the same output. This would make transitions between high and low resolution areas smoother and ultimately result in a better visual output. Due to currently unknown factors in RenderChimp this was not possible to try, an otherwise interesting hypothesis to test out. Figure 5.7 shows an output where an attempt to include an additional resolution buffer was tested.
Figure 5.7: The sponza scene with mixed resolutions. The black areas should have a resolution between the other two resolutions seen in the image.
5. Results
Chapter 6
Discussion, Conclusions and Future work

The result we got did not show any improvement in performance of the sponza scene, that is the FPS count was not improved by this kind of graphic rendering. There were a number of factors that should be taken into account as to why the result was not improved.

The first thing that slowed the speed down were all the OpenGL/GPU calls that was added in order for this code to work. Instead of having just one full resolution buffer rendered and added to the screen, we had a bunch of different kinds of buffers. One low resolution buffer, one high resolution buffer and one sobel buffer. Every time the scene was going to be rendered, these buffers were send through the shaders for calculations. Compare this to a single buffer with two single shaders and it becomes clear that the OpenGL calls lower the overall speed.

The way to improve this is to have lower number of OpenGL calls. To have more effective calls and merge some shaders to be more effective.

Another thing that slowed the program down, which was mentioned earlier, is that the full resolution buffer was always in use. Instead of only using it for rendering a selected area that was supposed to be high definition, it was rendered all the time. Although, because it was not writing to all the pixels on the screen, the speed was improved. The higher resolution buffer is an important part of the program since every other attempt to remove it or make it smaller failed. It only resulted in more calls for shared memory between the CPU and GPU.

We did try to work around the full screen resolution buffer by divided the scene into tiles and use glReadPixel. With that program we only rendered the full resolution screen at certain tiles but the glReadPixel call was the bottleneck. If we could have worked around the glReadPixel, maybe used a smaller buffer for the glReadPixel to read, it would have worked better. Though we did try to do the tiling in the shaders it is on the CPU at the end that will tell the GPU which resolution each different tiles must have. We saw in figure 5.5 and 5.6 that the full resolution buffer lower the speed of the performance even when it was not in use.
At the start of the thesis it was part of the plan to move beyond RenderChimp, to only use it at the start to get a general outline of the code. Because RenderChimp was a large program we had trouble to add bits of code. One issue was when we upgraded from the current OpenGL version used, 2.3, to the latest version, 4.3. Different libraries were added and it became troublesome setting the whole thing up.

On the plus side, our program worked fairly well in the effect to give a higher resolution to areas with details. When moving and looking around it followed the viewer and filled in the missing holes. The general idea of scenes being rendered like this is good, because when looking at a scene it is only the details that are interesting.
Bibliography


Appendices
Appendix A

Compute shader

```c
#define version 430
#define EXTRACT_DEPTH(cc)
   ((cc).b + (cc).g / 256.0 + (cc).r / (256.0 * 256.0) +
   (cc).a / (256.0 * 256.0 * 256.0))
#define TILE_W 16
#define TILE_H 16
#define OFFSET 2
#define BLOCK_W (TILE_W + OFFSET)
#define BLOCK_H (TILE_H + OFFSET)

const ivec2 tileSize = ivec2(TILE_W, TILE_H);
const ivec2 blockSize = ivec2(BLOCK_W, BLOCK_H);
const ivec2 filterOffset = ivec2(OFFSET/2);

uniform image2D output;
/* layout(binding = 0) */ uniform sampler2D normalBuffer;
/* layout(binding = 1) */ uniform sampler2D depthBuffer;
/* layout(binding = 2) */ uniform sampler2D diffuse8;
/* layout(binding = 3) */ uniform sampler2D diffuse16;
/* layout(binding = 4) */ uniform sampler2D diffuse;
uniform vec2 invRes;

void memorySync() { memoryBarrierShared(); barrier(); }
layout (local_size_x = TILE_W, local_size_y = TILE_H) in;

shared float pixel_depth [blockSize.x][blockSize.y];
shared float pixel_normal [blockSize.x][blockSize.y];
shared vec4 pixel_color [blockSize.x][blockSize.y];
shared vec4 pixel_color_16 [blockSize.x][blockSize.y];

void main()

```
A. Compute shader

```cpp
const ivec2 tile_xy = ivec2(gl_WorkGroupID);
const ivec2 thread_xy = ivec2(gl_LocalInvocationID);
const ivec2 pixel_xy = tile_xy * tileSize + thread_xy;
const uint x = thread_xy.x;
const uint y = thread_xy.y;
float tmp = 0.0;

vec3 normal = (texture2D(normalBuffer, (pixel_xy - filterOffset) * invRes).xyz) * 2.0 - 1.0;
vec3 pixelNormal = (texture2D(normalBuffer, pixel_xy * invRes).xyz) * 2.0 - 1.0;

/* Read pixels values blockwise (tile with apron), and store in shared matrices. */
/* Then sync all blocks with a memory sync. */
for (int j = 0; j < blockSize.y; j += tileSize.y){
    for (int i = 0; i < blockSize.x; i += tileSize.x){
        if (x+i < (blockSize.x) && y+j < (blockSize.y)){
            pixel_color[y+j][x+i] = (texture2D(diffuse, (pixel_xy + ivec2(i, j) - filterOffset) * invRes));
            pixel_color_16[y+j][x+i] = (texture2D(diffuse16, (pixel_xy + ivec2(i, j) - filterOffset) * invRes));
            pixel_depth[y+j][x+i] = EXTRACT_DEPTH(texture2D(depthBuffer, (pixel_xy + ivec2(i, j) - filterOffset) * invRes));
            if (cross(pixelNormal, normal) != vec3(0.0)) {
                tmp += 1.0;
                pixel_normal[y+j][x+i] = tmp;
            }
        }
    }
}
memorySync();

vec2 weight = vec2(50.0, 0.01);

/* Create Sobel operator matrices. */
mat3 sobel_x = mat3(vec3(1, 2, 1), vec3(0, 0, 0), vec3(-1, -2, -1));
mat3 sobel_y = mat3(vec3(1, 0, -1), vec3(2, 0, -2), vec3(1, 0, -1));
mat3 wxD = weight.x * sobel_x;
mat3 wyD = weight.x * sobel_y;
mat3 wxN = weight.y * sobel_x;
mat3 wyN = weight.y * sobel_y;

float sum = 0.0;
float sumX = 0.0;
float sumY = 0.0;
float sumXN = 0.0;
float sumYN = 0.0;

vec4 color = vec4(0);

/* Read stored pixel values from shared matrices, construct sums for depth and normal values. */
for (int i = 0; i <= OFFSET; ++i)
```
for (int j = 0; j <= OFFSET; ++j) {
    sumX += sobel_x[j][i] * pixel_depth[j][i];
    sumY += sobel_y[j][i] * pixel_depth[j][i];
    sum += sobel_x[j][i] * pixel_normal[j][i];
    sum += sobel_y[j][i] * pixel_normal[j][i];
}

sum += weight.x * sqrt(sumX * sumX + sumY * sumY);

/* Set output color to a specific resolution, depending on the value of sum and the threshold. */
color = sum >= 0.5 ? pixel_color[y][x] : pixel_color_16[y][x];
//color.rgb = sum > 0.5 ? vec3(1.0,1.0,1.0) : vec3(0);
imageStore(output, pixel_xy, color);
}
A. Compute shader
Appendix B
Sobel shader

```cpp
#define version 150

uniform sampler2D diffuseTexture;
uniform sampler2D depthBuffer;
uniform sampler2D normalBuffer;
uniform sampler2D lightBuf;
uniform vec2 invRes;

in vec2 tc;
out vec4 fColor;

#define EXTRACT_DEPTH(cc) ((cc).b + (cc).g / 256.0 + (cc).r / (256.0 * 256.0) + (cc).a / (256.0 * 256.0 * 256.0))

void main() {
    float cols[9];
    float colsDepth[9];
    float colsNormal[9];

    vec2 offset[9];

    vec3 pixelNormal = texture2D(normalBuffer, tc).rgb;
    pixelNormal = (pixelNormal * 2.0) - 1.0;
    int k = 0;

    for(int i = -1; i <= 1; i++)
        for(int j = -1; j <= 1; j++){
            offset[k] = vec2(j, i);
            k++;
        }
}
```
vec3 color;
vec4 depth;
vec3 normal;
vec4 light = texture2D(lightBuf, tc);

for (int i = 0; i < 9; i++) {
    cols[i] = texture2D(diffuseTexture, (gl_FragCoord.xy + offset[i]) * invRes).r;
    depth = texture2D(depthBuffer, (gl_FragCoord.xy + offset[i]) * invRes);
    colsDepth[i] = EXTRACT_DEPTH(depth);
    normal = texture2D(normalBuffer, tc + offset[i] * invRes).rgb;
    normal = normal * 2.0 - 1.0;
    if (cross(pixelNormal, normal) != vec3(0.0))
        colsNormal[i] = 1.0;
    else
        colsNormal[i] = 0.0;
}

mat3 Adepth = mat3(colsDepth[6], colsDepth[3], colsDepth[0],
                    colsDepth[7], colsDepth[4], colsDepth[1],
                    colsDepth[5], colsDepth[2]);
mat3 Anormal = mat3(colsNormal[6], colsNormal[3], colsNormal[0],
                    colsNormal[7], colsNormal[4], colsNormal[1],
                    colsNormal[5], colsNormal[2]);

mat3 Gx = mat3(vec3(1,2,1), vec3(0,0,0), vec3(-1,-2,-1));
mat3 Gy = mat3(vec3(1,0,-1), vec3(2,0,-2), vec3(1,0,-1));

mat3 AxD = Gx * Adepth;
mat3 AyD = Gy * Adepth;
mat3 AxN = Gx * Anormal;
mat3 AyN = Gy * Anormal;

float sum;
float sumxN = 0.0;
float sumyN = 0.0;
float sumxD = 0.0;
float sumyD = 0.0;

for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++){
        sumxN += AxN[i][j];
        sumyN += AyN[i][j];
        sumxD += AxD[i][j];
        sumyD += AyD[i][j];
    }
}

sum = 0.1 * sqrt(sumxN * sumxN + sumyN * sumyN);
sum += 0.1 * sqrt(sumxD * sumxD + sumyD * sumyD);

//sum += light.r + light.g + light.b;

if (sum > 0.2)
    fColor.rgb = vec3(1.0);
else

    fColor.rgb = vec3(0.0);

    fColor.a = 1.0;