Aaru‘s Awakening Special Effects

B.Sc. project for computer science

Fall 2013

Instructor: Birgir Kristmannsson
Examiner: Torfi H. Leifsson

Student: Marinó Vilhjálmsson
Partner: Lumenox ehf.

Contents

Abstract

The project is a set of three computer graphic special effects for the 2D platformer video game *Aaru’s Awakening,* currently being developed by the video game developers at *Lumenox*. This product will be used by the developers and art director at *Lumenox.* This can only be run inside the environment it was developed for, which is the *Unity3D* game development engine.

Of those three effects, only two were researched and implemented and though some research was put into the third, there was not enough time to finish and implement it. The two special effects that were implemented are *Shadow Cutout* and *Heat Distortion* for full screen background and selected geometry. The third was lighting for the main character’s teleportation projectile. Aaru is the game’s main character, and one of his special abilities is firing a burning projectile that he can then later teleport to, where ever it lands. The theory that was cooked up will be explained to some extent in this report to give some idea on how someone could implement it.

The architecture of the video game which will use these effects, requires specifically designed effects for various reasons. The game is in 2D as a result the geometry has no way of describing the way it should react to lighting, secondly the art was not created with surface describing maps from the start and testing of some other general solutions had already been fruitless. The methods that were researched are explained in detail.

Graphic programming requires that the result has to look as good as possible for the smallest computing effort possible, and part of the problem is finding the correct scale of good enough and fast enough.

Even though these effects were developed for *Aaru’s Awakening*, they were designed to be loosely coupled and can be used on similarly constructed environments. The theories and methods of the effects are not platform specific, and can be ported to other projects.

Brief description of shaders

Shaders are programs that are run by the *GPU (graphic processing unit),* shaders are generally simple methods that describe how computer graphics are displayed to the user. Shaders can be split into three categories for computer graphics, though some additional uses for them have risen up for other calculations which will not be mentioned further here.

Shaders can be split into *Vertex-*, *Fragment-* and *Geometry shader* programs. Geometry shaders are relatively new in computer graphics and will not be discussed here since they are not relevant to the project.

When a 3D environment is built it has an abstract point in space to represent the absolute zero point in the environment referred to as *World Space*, and to display that environment to the monitor there will have to be a viewing direction and position in space to represent the observer and describe what angle the scene is viewed from, this observer is referred to as the camera.

A very basic 3D/2D model is described by points in space, along with simple information on in what order to connect them. When a scene is drawn to the monitor, only a set of objects are *“visible”* to the camera and those objects need to be transferred from their world space into the 2D space of the monitor *(screen space)*. That’s where the shaders come into play if, that is if we omit the *Geometry Shader*. A *vertex shader’s* job is to do the calculations required to map the object from world space into screen space, describe how to draw *textures* onto the geometry and other interesting calculations, and then feed the information to the *fragment shader.*

Finally the *fragment shader* connects the dots and actually handles drawing the geometry using the information from the vertex shader and the program itself, such as *textures*, *maps* and other parameters. The *fragment shader* handles the coloring of the pixels bounded by the vertices of the 3D model, the pixel’s transparency and how they react to light sources in the scene.

For a scene there can be a lot of geometry and though various methods are used to reduce the number of calls to the shader programs, such as not drawing things that are occluded by other objects or not inside the view of the camera, there is still the bare minimum of calls to the fragment shader of each pixel of the display resolution, which is a few hundred thousand pixels *(around 2.3 million pixels for High Definition)* on an average computer screen per frame where each frame has to be drawn at least 30 times per second to maintain a smooth display of movement and a pleasurable interaction with the 3D world. In other words, shaders need to be as compact and have as few instructions as possible and they are massively parallel so they cannot know about the results of other shaders in the same frame, so more often than not you have to be satisfied with “*good enough”* in real time graphics.

Height-maps are often mentioned in this report, and only relevant to explain them. A height-map is just a single color image, where each pixel of the image represents height, for a black and white height-map white could represent the highest point and black the lowest, and every point of gray in between linearly describes the height between black and white.

Introduction

This project is a custom asset package for the *Unity3D game development engine*. The final product will be a simple to import *asset bundle* with minimal ties into the application it is designed for. It was designed to be loosely coupled to begin with but there was a small part that was taken into account by designing these special effects specifically for *Aaru's Awakening.* Their design of the game world was one of the main reasons that *Lumenox* could not use general solutions of special effects for their game.

By minimizing the ties to the structure of the game itself, the special effects can be used for other games designed around using the same framework as *Aaru's Awakening* or even games that share the similar structure of layer based environment drawn by multiple different cameras and the theories can be ported into other project environments as well.

Difficulties

From the start one of the students in the group had substantially less time than the other, which resulted in delayed development, soon after starting the project the other member had to move to another city and continuing the project was done through Skype sessions and shared desktop. This surely slowed down the project and meeting time deadlines failed. Then unavoidably half way into the project he had to quit entirely which further slowed down the process. He tried his best to continue, but unforeseen complications forced him to quit the project. Resulting in a substantial part of the project being done by one student.

Environment

Coding

The project was developed mostly in C# and the shader language CG. Though the project can probably be compiled outside of Unity3D it wouldn't make much sense to do so. Code was mostly written using the *MonoDevelop* ide, but later got migrated into *Visual Studio* for that environment was much more effective at reading the built in DLL's from *Unity3D* and gave a smoother and less quirky development pace.

CG is written mostly in a text editor of choice, *Sublime Text 3*. The base of the project is structured around overwriting the *Unity3D MonoBehavior* class's default behavior, which is the abstract class for handling calls to the user generated scripts at various points in the game loop. MonoBehavior handles events such as collisions, objects entering or leaving the camera's view space, the event of destroying or creating objects and a lot of other useful methods were and could be useful.

Additionally there were implemented extended methods for the *Unity3D development editor* which will give the developers an interface between the effects and the game engine, an intuitive way of creating and managing the objects in the scene.

Testing

For this particular project testing is hard to automate given the time scope of the project, testing was performed manually for those particular parts with simple helper scripts and visual testing was also a large factor *"if it doesn't look good, it isn't good"*.

Another vital part of testing for computer graphics is performance testing. Considerable time went into rearranging code and figuring out intuitive ways of minimizing unnecessary function calls and shader calculations which could be administered by monitoring the frame rate of the game, (how many frames are drawn per second). The project was not very large so exhaustive manual unit testing was reasonably feasible. Unity also has a built in profiler, which greatly helped performance testing and finding bottlenecks.

Documentation

The documentation is done with *DoxyGen,* which is a free automated documentation generator for various programming languages including C#. It can output in various formats and the format of choice for this project was html, which is easy to navigate and can easily be put online.

Researched methods

Shadow cutout

Shadow cutout is an effect that is used to mimic lights being visible in darkness, the reason we don't use traditional lighting is because of the architecture of the game environment. The game is designed with self- illuminated 2D objects, which is chosen to better control the look and feel of the environment.

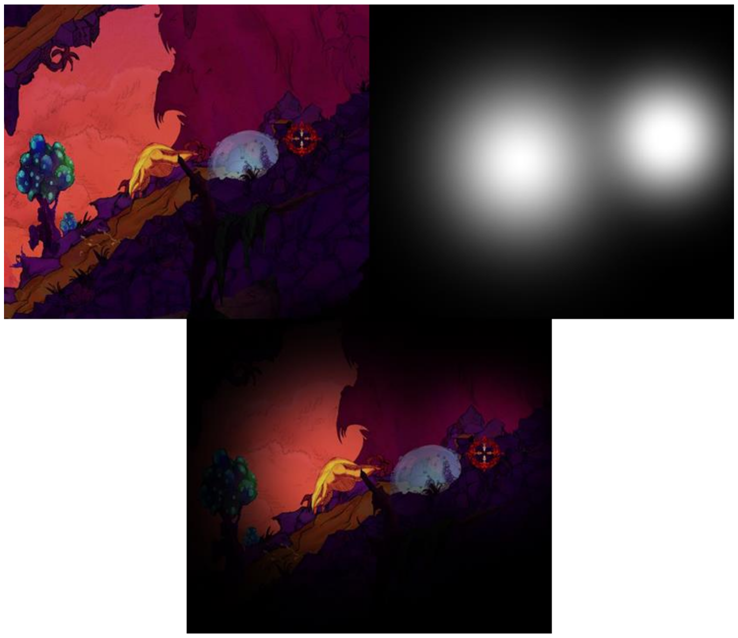
FIGURE 1 SHADOW CUTOUT MOCK

Idea

The idea behind the shadow cutout is based on a simple method of stencil masking. Cut a hole in a black sheet where the object is located, and place the sheet over the image like shown on the picture to the right. The stencil mask is drawn by iteratively cutting out an alpha mask from the stencil for each object that is marked as a light source. And finally drawing it over the scene.

Method

This effect is used by attaching a script to the camera that should display the effect and primitives that hold alpha maps (*"stencil holes"*) that should be their cutout placed on objects that will act as light sources. The script on the camera creates another duplicate of the camera it's attached to, and changes its culling mask so it can only see the primitives that hold the alpha maps. Then the duplicate camera will draw what it sees into a render target and filling up the space where no objects *("lights"*) were drawn with a black opaque hue. Then the game goes on its merry way and draws what the original camera sees into the default render target. When we have both the original frame, and the stencil map that was drawn before, it uses a full screen shader to blend them together into one final frame to send down the road to the next camera in the scene or to the final color buffer to display on the screen.

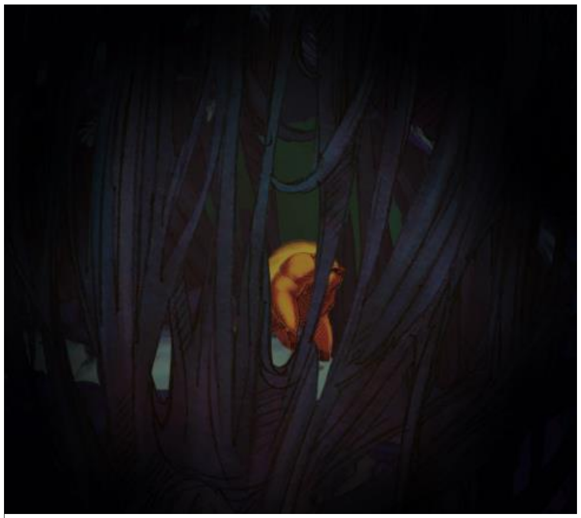
FIGURE 3 LIGHT SHINING THROUGH THE FOREGROUND

One problem rose up with this method, the effect makes no distinction between what parts of the environment that are being drawn by the camera whether they are in front of the light or behind it, this resulted in the light *„shining"* through objects that should have occluded the light as can be seen in (Figure 3). This problem had to be fixed and annoyingly enough there was no possible solution found for this that did not have to make changes to the environment, so *Lumenox* allowed changes to be made to make a distinction between the foreground and background by dividing them into two separate layers and tell the camera to render them both. With that new layer the temporary effect camera could render the foreground separately and use it to mask out parts of the stencil map. So the foreground color was used as an alpha value, resulting in a nicer effect (Figure 2). This can be toggled on or off for each light, if for some reason the light should shine on the foreground, it can easily be toggled.

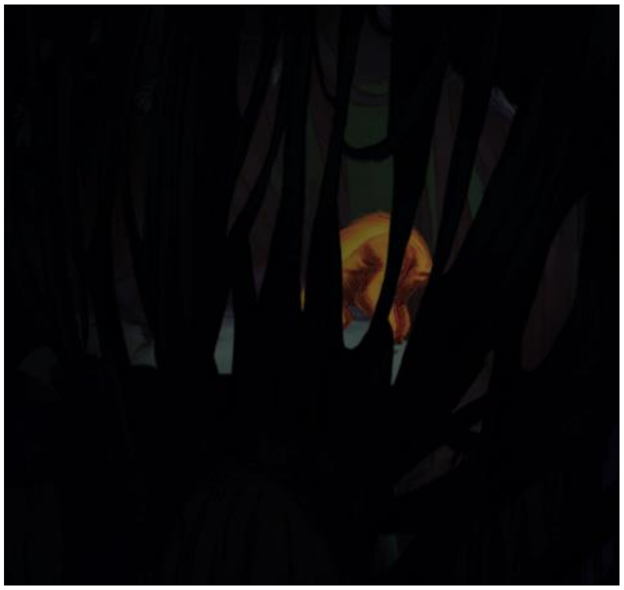
FIGURE 2 AFTER STENCIL MASKING

Another method that was tried was doing this entirely a full-screen effect, thus not putting any primitives to hold the alpha maps for each object, but instead only locations were sent to the shader about where to cut out. This worked to a point where the program started hitting obstacles and experiencing limitations. There was a considerable amount of time spent on this method that was wasted, it had to be canceled and sent back to the drawing board. The most inhibiting limitations were complex animations and light blending (Figure 4), also it was not performing well enough when multiple lights were in the scene. It was too big of an impact on overall frame rate.

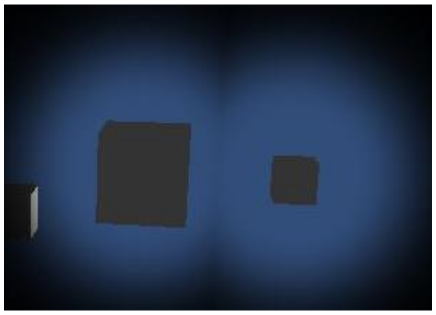
FIGURE 4 BLENDING PROBLEMS

Further study

A request that one of the *Lumenox's* artist brought up was making the light diminish as the background was further away, but this request came a little bit too late so it was not implemented. There was put a little bit of effort into thinking about how to do that but did it was not implemented.

The plan was to write a shader that could draw the depth (*the z distance from the origin of the screen*) of the objects rendered by the cameras behind the one with the shadow cutout effect, and use that information to scale down the intensity of the light, the higher the depth the lower the intensity of the light on that point. This is a little more effort than simply reading from the depth buffer at that given time in the render loop, since most of the objects in the game are simple quad primitives with alpha blending, so no useful depth information is written by default and the depth information would have to be manually alpha blended and calculated with so called "*replacement shaders*" which render the whole scene with a different set of shaders for every piece of geometry.

Heat distortion

Heat distortion is the phenomenon that happens when light chooses a shorter path toward the eye of the viewer. The reason light chooses this different path is when air heats up it becomes less dense, thus allowing light to pass through it at a greater speed. So when something that is particularly hot it will heat up the air around it causing it to "bend" the light towards it, creating the illusion of seeing things as displaced from where they actually are. Like mirages in the desert are caused by the heat of the air just above the surface of the sand to "pull" the light towards that is being emitted just above the horizon and thus creating the illusion of a water surface reflection, where you are technically just looking at the sky just above the horizon. These features would be very difficult to mimic with real-time rendering 3D engines, but we can "sort of" get the effect by making textures that tell the 3D renderer how to offset the pixels that are finally displayed on screen.

Heat distortion is used in the game's hotter zones and is used to give the areas that are placed in the game world *"Day"* some life by distorting the background in the distance and heat distortion over pools of lava and tar. The heat distortion effect is twofold, both full-screen and heat distortion to set on geometry that can be alpha masked. The full-screen effect is used on a selected camera to distort everything it, and previously rendered cameras see, e.g. the background. The heat distortion for the lava and tar pools would have to be placed on quads, circles or other simple primitives, it can be alpha masked to make it blend into the environment and shape it around complex forms. Since there can be a considerable amount of hot spot the shader has to be more optimized than its full screen counterpart.

Idea

The idea behind the heat distortion is simple, take the image to be distorted, and run it through a filter that offsets the pixels based on vectors defined on a texture map. The vector map, also referred to as an offset map and is generated from a height map. The idea is to run the rendered game through this distortion map and have the offset map moving in the same direction the hot air is supposed to be moving.

The offset map is based on a well-known method of normal mapping. Where a normal map describes the surface of geometry by holding 3D vectors for each pixel or a set of pixels to describe the orientation of the surface. So using a similar method but to store 2D vectors in the offset map that will describe in what direction and how far to sample a pixel of the image that is being distorted and display it (the theory of using offset maps is not original, and has been used in the distortion of reflections of a moving water surface). Both shaders are built around the same theory but are different in scope.

Method

The format of the offset maps is pretty simple, it uses the red and green color channels of the texture for the offset vector. Texture colors are in the range 0.0 - 1.0, using the red and green color channels of the texture, where 0 is mapped to be -1 and 1 to be +1 on the X and Y coordinate respectively. Where the -1 to 1 on the red channel describes horizontal offsets, the greens describe the vertical offsets, these vectors are then scaled up by a factor adjustable in the Unity3D editor.

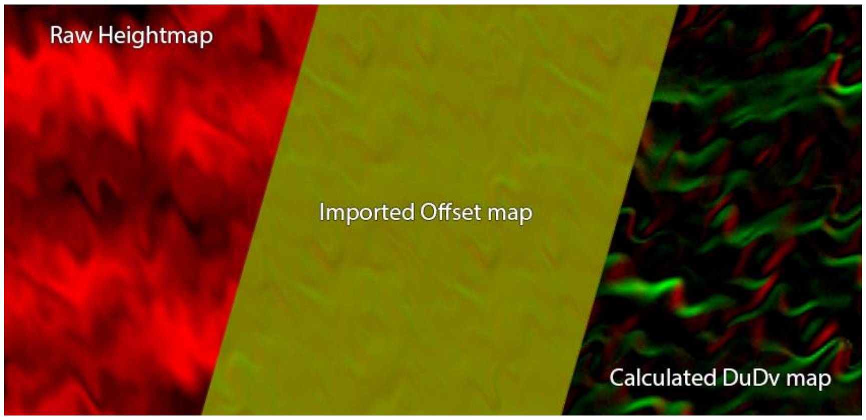

FIGURE 5 THE VARIOUS STAGES OF THE OFFSET MAP

Making these textures is not arbitrary, they have to be calculated from so called height maps. Height maps are really just a single-color texture that has some sort of pattern, be it a noise pattern or some form of waves, is something to experiment with. The approach that was taken was to overwrite the asset importing method for textures, and applying that overwrite for all textures that will be imported into a certain folder of the project. The method reads each pixel of the height map, and takes the difference in color value (height) of itself and the adjacent pixel on X and Y axis. These values are in the range -1 to 1 and need to be mapped into the range 0-1 which the texture can hold. Through some research it was found that it's not consistent between gaming platforms on whether textures can contain negative values or not, so the approach was taken to map the values to the 0-1 range and back to -1-1 in the shader program. As can be observed in the image, the texture starts out as a red height-map with an arbitrary pattern of some sort (the reason red was picked will be discussed in the further study section). Then the imported Offset map is when the difference in "*height*" has been calculated. And finally the calculated map, the shader actually uses to offset the UV coordinates, note that black is actually either no offset at all, or it is a negative value (an offset left or down), since colors are represented as 0-1, there is no way to describe a negative color other than black.

Converting the height-map to the offset map as done by the importer:

$$offset.red_{ij} = 0.5 * [(height.red_{ij} - height.red_{i+1\,j}) + 1.0]$$

$$offset.green_{ij} = 0.5 * [(height.red_{ij} - height.red_{i\,j+1}) + 1.0]$$

Then converting the values in the offset map into offset vectors as done by the shader:

$$vector.x = scale * [(offset.red_{ij} * 2.0) - 1.0]$$

$$vector.y = scale * [(offset.green_{ij} * 2.0) - 1.0]$$

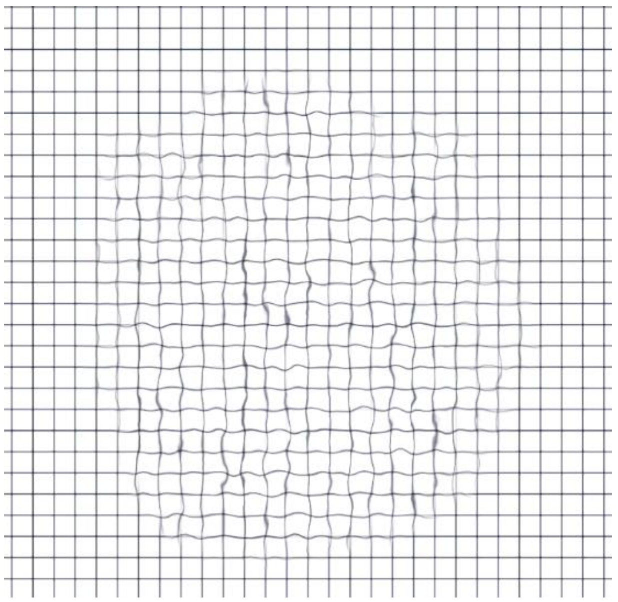
FIGURE 6 HOTSPOT SHADER WITH NOISE MAP AND ALPHA MASKED

For the *"hotspot"* shader there was used a single offset map that was moving in the positive Y direction, the offset map can also move on the X axis by a cosine function to give it a little variance but this is adjustable, it didn't always look good so the option to customize it is available. The method used to get the background image was using a shader grab pass which is an inexpensive way to get the current state of the current color-buffer. The first grab pass will generate the texture of the background and save it in a texture register on the GPU, and should a grab pass be called in a consecutive shader it will reference the previous grab pass, thus only the first grab pass will generate a texture, the next *n* grab passes will only reference the first. With this there could easily be a lot of hotspots without dragging the frame-rate down but by a tiny amount.

The full-screen effect was a little trickier, being a full-screen effect means it's bound to the movement of the camera, but that is not what was desired, the effect should be stuck to the background. For this it was necessary to *"wrap"* the offset map around the screen for both X and Y axis, and follow the background while at the same time moving the offset-map in the animation direction of the heat distortion. This proved to be very problematic and was among the reasons *Lumenox* could not use a general solution for this effect. The problem was further complicated when the camera was zoomed in and out; when the camera zoomed the scaling of the offset map changed relative to the background resulting in unwanted *"distorting movements"* this had to be compensated for.

The effect works pretty similarly to the geometry heat distortion only with more stuff going on. For this effect there are 2 offset maps, each moving on the Y axis independently from the other. Each map is sampled and then the vectors are linearly interpolated to make the final offset vector to offset the actual pixel. It was tested to make the linear interpolation run on a sine function to give the maps varying strengths toward the offset of the pixel, but in the end, a half and half split worked the best, animating between the maps resulted in noticeable patterns thus it was omitted and is a case for further development.

Further study

There were two things that were looked at to some extent, but not implemented. Firstly the animating of the offset maps inside the shader, like briefly mentioned in the full-screen effect where the linear interpolation of offset maps was animated. It is possible to make the interpolation follow a random noise map which could be the third texture map into the shader, calculating random numbers for each pixel in the pixel shader is a non-trivial matter and is often handled with noise-maps.

The other was a heat distortion effect specifically for *Aaru's* teleportation projectile. (*Aaru* is the main character of the game *Aaru's Awakening*, and his special ability is to fire a burning projectile that he can then later teleport to). There can be a method of doing the texture offsets as a sine-function of the pixels distance from the objects center and time instead of a baked offset map, then multiply the difference in the coordinate of the current pixel and the objects center, creating an offset vector that always points away from the object in the direction of the pixel, and scaling it by a value ranging from [-x - x *some sine function*] giving it a *"water pond"* ripple effect around the object.

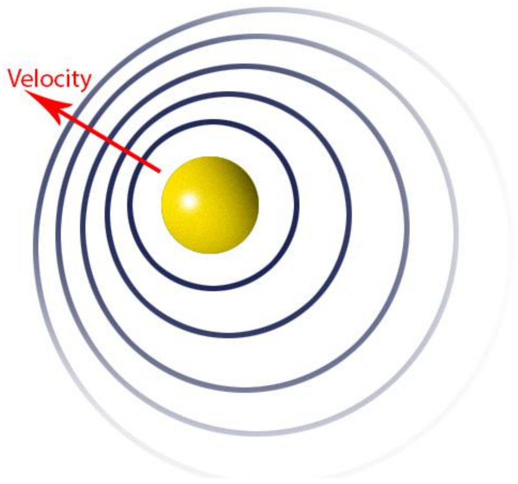

FIGURE 7 DEMONSTRATION OF HOW THE RIPPLES WOULD COMPRESS AT THE VELOCITY VECTOR

This effect would be placed on a primitive that would be rotationally locked and would follow the projectile. The matter of alpha blending the effect, this could be a function of the distance from the objects center. And finally in addition to the pond ripples around the object, the velocity vector of the projectile could be passed down into the shader to make the frequency in the travelled distance of the projectile higher, and thus less frequent in the negative velocity direction while making the alpha blending more aggressive in the direction of the velocity. This method never got any further than being an idea and there is nothing to show for it, but this would be a nice addition to the heat distortion effect library and given the information above, the effect should be reasonably easy to implement.

As was previously mentioned the red channel of the texture was chosen because; at first the alpha mask could be compressed within the blue channel of the offset map, the artist at *Lumenox* requested that this would be separated into an alpha texture of its own, so the blue channel is no longer used as the alpha map. Using the blue channel will reduce the number of texture samplers the GPU has to store, this increasing the performance of the shader, this was an optimization step.

Light source method

The light source was to be focused on the teleportation projectile of the game character *Aaru*. The problem with the current lightning of the projectile is that the geometry around the projectile has no real surface to interact with the light emitting from the ball, this making the lighting very flat and lifeless.

Idea

A fair amount of time was invested in figuring out a method for lighting, as mentioned above the game world has no surface to describe how it should react to the light of the projectile, but that does not mean it cannot be created. Bump mapping is a method used in 3D graphics to describe a complex surface on simple geometry by providing the light source information on how it should react with the surface of the object it shines on. As can be observed in (Figure 8), the sphere on the right looks like it a much more complex geometry but in truth it is does not, the only difference in those two spheres is the way they react to the light shining at them. The one on the left has only the geometry (the normal vector of the face) to indicate the orientation of the pixel the shader is calculating while the other one has a normal-map where the orientation of the pixel being calculated is not governed by the geometry but the normal-map instead, so the normal of each pixel is pre computed and stored in a texture as 3D vectors.

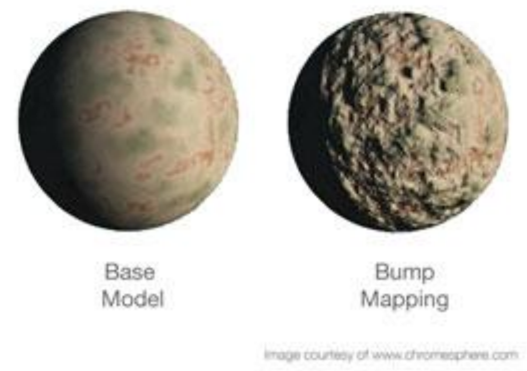

FIGURE 8 DIFFERENCE ON SIMPLE GEOMETRY WITH AND WITHOUT BUMP MAPPING

The way these vectors are calculated is from a height-map, similarly to the offset mapping described previously in this report, the vector is calculated from adjacent pixels. Normal mapping is done by taking the pixels to the left and right and creating a vector between them, then another vector is created from the pixels above and below and use the color value of the difference for the *z* component of the vector, and pretending x and y are 1.0 and 0.1 respectively. Now with these two vectors you should be able to figure out that they form a plane relatively tangent to the surface, then to generate the normal vector of the pixel, the cross product is taken of the two calculated vectors. This gives us a vector that is perpendicular to both tangents of the slope, or the "normal". These normal maps are usually pre calculated and generated from modified versions of the original textures that are on the geometry or generated from more complex versions of the geometry specifically made for the purpose of generating normals maps.

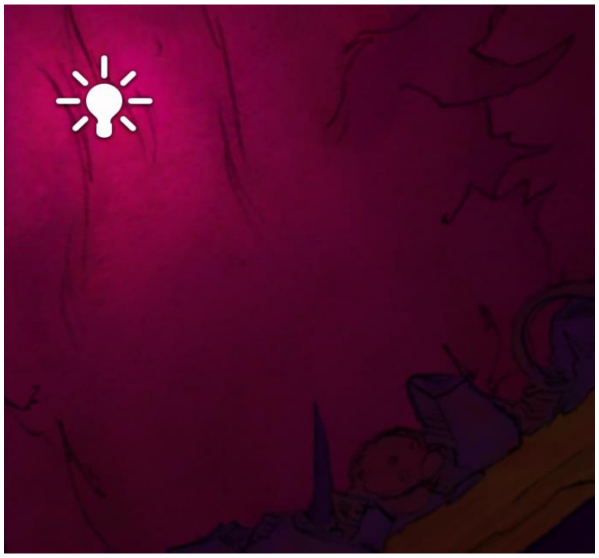
FIGURE 9 ORIGINAL TEXTURE

But another cheap way to make normal maps is to approximate the bumpiness of a texture by simply converting the texture into a grayscale image and pretending that it is the height-map for the texture and calculate the normal vectors. That is what was done for this example here (Figure 10) from the original texture (Figure 9). Since the task was to use the effect on the teleportation projectile mainly or only, it was not completely out of the picture to simply approximate the normals around the immediate location of the projectile. The figure (Figure 11) is an approximation of what the effect might look like should it be implemented.

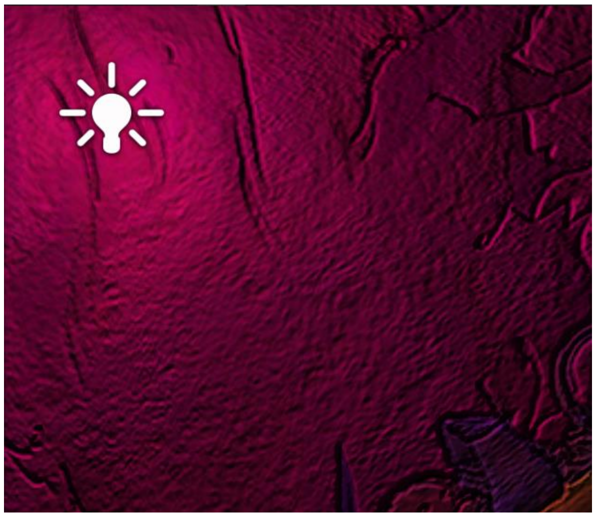
FIGURE 10 APPROXIMATED NORMAL MAP

This way the lighting would no longer be flat, since the immediate background of the projectile would be lit up and would interact with the light depending on the location of the light, the shading in crevices and nooks would respond to the movement of the projectile.

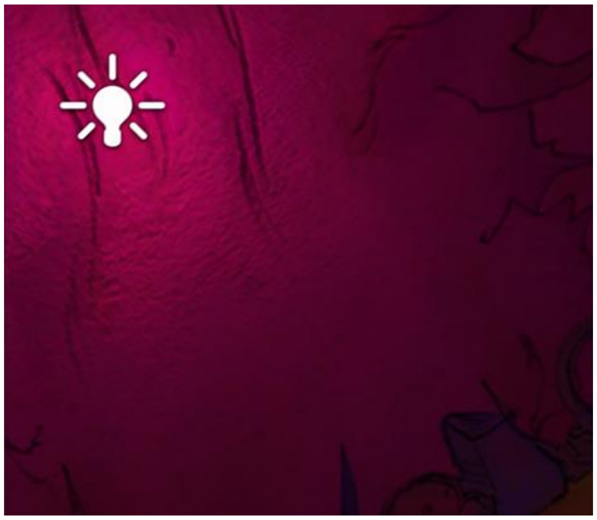
FIGURE 11 COMBINED PREVIEW OF THE EFFECT

As mentioned, this is only an idea, it was not implemented nor brought into any kind of testing within the environment of the game. The final result might not harmonize with the "*paint brushy*" feel of the overall art in the game, or just not look right overall. The concept looks promising though and could be a nice addition to the game. The shader would be a variation of the typical *lambert* lighting shader with the method of creating the normals on the fly instead of reading the normals from the geometry, in addition the lambert shading could be simplified even further since the direction of the camera is always constant, looking down the Z axis, and then finally alpha blend the results into the final draw making the approximated bumpiness less prominent as the light is further away until it fades into the original art of the textures.

$$normal_{ij} = normalize[(1, 0, pixel_{i-1j} - pixel_{i+1j}) \times (0, 1, pixel_{ij-1} - pixel_{ij+1})]$$

Project plan

The project does not follow any methodology like Scrum or Kanban to the letter. There were very vague information about the tasks and difficulties of the project. It was also difficult to generate good "*user*" stories and use cases. There was a guiding light of the final results that were desirable and there had to be some research on how to achieve that result creating tasks based on what was foreseen having to be done.

Working methods

The project was split into three standalone parts, the *Shadow cutout*, *Heat Distortion* and *Projectile light Source*.

Each segment should be taken and developed into a final piece before continuing onto the next. Each segment was split into subsections:

- Research
- Prototyping
- Developing
- Optimizing
- Testing

The original plan was to take three weeks into each primary part and spending the final part of the time documenting, writing user manuals, final report and integrating it as an easy to deploy import package to *Unity*.

First there was research of viable methods, when some methods were found that were plausible to work, they were prototyped into a simple (*dummy / mock)* environment. When the method of choice was found it was further developed into the project and made to work with the game environment provided by *Lumenox*. Then finally when the results were acceptable, they were profiled and ways to optimize were explored and implemented where possible. Then finally when the effect was ready it was tested, of course testing would be done gradually, but an extensive code breaking tests will made in the end, trying to find bugs however possible.

After research and prototyping we planned working together by pair programming when applicable, when programming of the effect was well under way the plan was to have the other group member start looking for plausible methods for the next effect and get the grounds covered so we could start prototyping faster after the first sprint. But as described above, this was only true for the very first part of the project as the other student had to quit the project entirely.

The way I managed after he quit, was to reorganize the project, cut out of it unnecessary parts in cooperation with the contact at *Lumenox* and only deliver what I foresaw that I could manage to complete. As a result the project was cut down in scope, only the first two effects would be completed, the shadow cutout and heat distortion. I did my best at following the original plan but everything happens slower with only a one pair of hands.

Project progress

There was not done a good enough job on bookkeeping the time and overall progress of each milestone and this definitely proved a problem. There are no good reasons for this other than being a major oversight and a possible result of under-planning. As a result of this; detailed sprint backlogs are not available, and a written description of the project progress will have to do.

First milestone: 15.09.13 -
Get everything ready to start research and development.

Here everything was completed in due time and the first part of the next milestone was started. It included research on methods for the shadow cutout and creating mock tests. A wireframe of what needed to be completed and when. Though later this proved to be too hastily done and planning needed to be refined.

Second milestone: 26.09.13 -
Have the static shadow cutout ready in the game environment with simple functionality.

Here everything went according to plan, the simple methods were all implemented and even material designed for the next milestone had been started.

The *Aaru's Awakening* test project was ready to be used and tested on. Considerable time went into getting the grounds on how the project was built and the overall structure of the layout of the project, this though went smoothly with a little help from the guys at Lumenox.

The simple functions like static light cutouts and function to apply the overlay map were applied on the actual game environment.

But here the impact of hasty planning was obvious, as new tasks were being invented during mid-sprint that could have been planned beforehand. This obviously produced problems with meeting deadlines.

Third milestone: 7.10.13 -
The shadow cutout effect, testing and documentation completed.

Here the three week deadline for the shadow cutout was at hand. The effect was far from finished. A lot a lot of problems rose up with the method of drawing the alpha-map for the effect that was chosen, and that had to be taken back to the drawing boards and rewritten from scratch, this was partly because of lack of organization, not complete enough requirements analysis causing new tasks to be invented mid sprint. But a large factor was also poor results from performance testing that did not meet the criteria that were set.

In addition the scale of the group’s time imbalance was starting to affect the development of the project. Documentation had not started and a lot of tasks were unfinished. Here it was unavoidable to make a new milestone to compensate for lost time.

The functionality for static and basic curve animated light cutouts was implemented, the effect could be tossed on moving objects, the lights could be colored, basic results of cutting the immediate foreground out of the lights to prevent light shining through caves, rocks and vegetation was implemented.

Fourth milestone: 21.10.13 -
The shadow cutout effect, testing and documentation completed “refined”.

This milestone was created to compensate for the amount of work that remained on the shadow cutout effect, at this point some work was left on documentation and testing, which was taking considerably longer than planned. The state of the effect was complete though a tiny amount of functionality would be added later at the end of the project to make the effect simpler to use within the game editor. Performance testing results looked promising.

Here most of the functionality of the effect was working, a new method of drawing the alpha map in one shader pass had been implemented. Sprite animations had been implemented with various functions to control animation states. Curve animations refined to work with the new method of drawing the alpha maps, now utilizing the Unity3D graphical curve creation tool for more complex curve functions. Foreground stencil cutting was completed and the option to choose which shader to use. The default GUI inspector for the scripts had been overwritten to give control of animation curves, sprite sets and overall govern the access to parameters of the effect. The ability to disable and re-enable the effect during runtime had been implemented.

Profiling for bottlenecks had started along with writing of user guides.

Fifth milestone: ~~1.11.13~~ - 5.11.13 -
The heat distortion effect mocked and ported into the game environment.

This milestone was drawn back by four days to give time enough to complete what remained of the shadow cutout documentation, testing and complications with one team member quitting the project. At the start of this milestone the planning for the effect was revised to prevent screw ups like in the previous sprints. A meeting with *Lumenox* was held and requirements were made more concrete than in the previous effect, the planning was revised so to not underestimate the time taken for testing, and additional time given to compensate for the project would be finished by only one group member.

Research on methods and a mock version of the heat distortion effect had been created. A custom texture importer to generate offset maps from height-maps had been created, and work toward implementing the effects into the game environment was started.

Sixth milestone: 25.11.13 - The heat distortion effect, testing and documentation completed.

Here mostly everything was completed as planned, though some of the documentation remained to be finished. Reforming of the requirements and greater planning helped greatly in preventing new unplanned tasks to be added to the project like previously, and overall gave a clearer image on what needed to be done. What remained of the documentation would be finished in the next sprint where the project is boiled together to be ready for deployment and hand-in.

The full-screen and hotspot effects were implemented, profiling for bottlenecks was done, along with optimization where possible. The inspector GUI for the hotspots was overwritten to allow live modification of the effect in the editor. Methods to minimize the calling of effects that are not visible on screen were created. User guide was almost completed.

Seventh milestone: 9.12.13 - Finishing up the effects and make them a deliverable component.

Here the documentation was finished, code was refined and bug-fixed and necessary scripts added to the project to make it easier to use within the Unity3D editor.

The functionality for the context menus was added to create effects or add them to previously created objects. Minor issues with managing the folder hierarchy was resolved so the project folder can be anywhere within the assets of the Unity3D project. Documentation was finished and generated with DoxyGen. Final bug searching and fixing done, some minor bugs fixed.

Retrospect

There was definitely lack of organization in the project. Programming was started too soon, which caused new tasks to be created halfway into the sprint. Time required for testing was greatly underestimated, the tests had to be administered manually. Visual testing is a non-exhaustive task and finding the point where one could be happy and effectively stop proved to be difficult and time consuming if not careful. The group’s time investment ratio and the one of the group member quitting was not something that could have been planned or prevented so it might not be right to mark that as something that could have been differently, though it undoubtedly had an impact on the project.

A lot was learned on the Unity3D environment as the project was being developed and solutions to problems that were previously faced could have been handled differently. In retrospect I am not entirely happy with the overall structure of the Shadow Cutout effect, and believe it should have been broken down into an interface and 3 implementations, one for each type of animation type, this is a direct result of the lack of planning at the start of the project.

Finished product- what was delivered

The start of the project was designed around three effects, but complications resulted in only two of them being implemented. The definition of the project request stated that only two of the effects would be adequate but three would be desirable. The effects *shadow cutout* and *heat distortion* where the main requirement and due to complications of the group, that was only what was delivered.

The shadow cutout effect with static lights and two variations of animation possibilities completed with documentation along with the heat distortion for background and selective geometry along with documentation. For both effects the required guides to import them into the game environment of *Aaru's Awakening* were provided, along with the instructions on what changes were needed to be made on the architecture of the game to have certain options of the effects available. Though some height-maps for the heat distortion are provided, the format and pattern of the noise in the maps is still a case that can be studied and tested further to get better looking results. Instructions on how to make new maps is also provided in the *Researched methods* chapter.

Methods to implement the creation of *"effect objects, e.g. hotspot, cutout source"* were added to the *Unity3D* context menu for ease of use. Creating effects from the context menu creates the necessary game objects to hold the effect, and configure them to the default settings. And the shadow cutout the effect can be added to existing object or a list objects with the context menu. It is not necessary to create the effects using the context menu. They can also be created manually by placing the correct scripts on the game object of choice following the description in the provided user guide. For each effect a custom "*Unity3D inspector GUI" was* created, providing access to parameters so that can be modified in an intuitive manner in the editor, while also constricting access to parameters that should not be tempered with given the effect's state, this can be circumvented by changing these values through other scripts, but it's recommended not to do so unless necessary.

Both effects were performance tested and neither crossed the boundary of performance cost for a complex scene (which was not concrete, but a mutual arrangement was at around 25% impact on frame-rate) with multiple instances of the effects at play. Both are fully functional with shader model 2.0c and above.

The effects can easily be turned on and off during runtime, they handle disabling necessary scripts and threads themselves, so the developer does not need to worry about dangling threads or unwanted function calls to remain in the game loop when he disables them. The effects' scripts can be communicated with by other scripts in the scene which gives flexibility to modify the parameters of the effects with events of the game while it is running. Effects can be instantiated during runtime of the game and can be fully configured through calling scripts of other objects. The effects can be attached to most logical game objects. There remained an issue with *Shadow Cutout Curve animations* where strange artifacts were observed when disabling animations on camera culling, this problem was not fixed and currently the curve animations still run when they are note seen, time could not be found to rectify this.

The third effect specific lighting could not be completed, though considerable thought and planning on how it could be designed and implemented, so a comprehensive guide on how to implement it is provided in this report under *Researched methods – Light source method* chapter. Additionally some thought on how to improve the two completed effects are provided in this report under the *Researched methods*

What was learned

I learned a significant amount about the inner workings of the *Unity3D* and the flexibility of extension methods for *Unity3D.* I also greatly strengthened my knowledge of computer graphics in general, the state of shaders today, and various ways you can use texture maps to bake calculations to save time during runtime. I learned that the usage of shaders is a rapidly growing field and it's not only usable for computer graphics, but is increasingly being used for more and more computations in various fields. *Unity* offers the possibility of computation shaders with the *DirectX 11* library. Computation shaders are then used for calculation of various other tasks than graphic related.

Good organization and planning is something that can always be improved and should be given great care to make it serve its purpose of being useful. Something gradually built like a program requires careful planning, strict rules, an organized architecture and a good foundation or it gets chaotic and out of hand very fast, similarly as was encountered at the beginning of this project.

References

As mentioned the project was done mostly by Marinó Vilhjálmsson, but the project was started along with Ragnar Vilhjálmsson, who also helped develop and research some of the methods that were used for the project. There was extensive use of the Unity3D's great documentation, along with unity answers reading over old questions and answers, also the Nvidia's shader manual. Also overall google was used considerably.

Unity script reference: http://docs.unity3d.com/Documentation/ScriptReference/
Nvidia's CG shader manual: http://developer.download.nvidia.com/cg/Cg_3.0/CgUsersManual.pdf