Alastair Toft

Fast, accurate reflections with parallax-corrected cubemaps

Computer Science Tripos – Part II

Downing College

May 19, 2017

Proforma

Name:	**Alastair Toft**
College:	**Downing College**
Project Title:	**Fast, accurate reflections with parallax-corrected cubemaps**
Examination:	**Computer Science Tripos – Part II, July 2017**
Word Count:	11,532
Project Originator:	Huw Bowles
Supervisor:	Gyorgy Denes

Original Aims of the Project

Fast and realistic rendering of reflective surfaces is a challenging problem in real-time 3D graphics. This project aims to implement a number of existing approaches to the problem, and to implement a novel technique for rendering reflections using cubemaps and depth textures. The goal is to evaluate the relative advantages and drawbacks of the new and existing techniques, based on their performance and visual results.

Work Completed

I have implemented techniques for cubemap reflections, parallax-corrected cubemap reflections with proxy geometry, and a new technique using ray-marching. I have implemented the Planar reflections technique and a CPU ray-tracer for comparison. I have integrated these techniques into the Unity game engine and its Editor. I have quantitatively and qualitatively evaluated each of these techniques.

Special Difficulties

None.

Contents

Chapter 1

Introduction

In this project I have implemented a range of techniques for producing reflections in real-time, including the implementation of a new technique for computationally cheap, accurate reflections. I have integrated these techniques into the Unity game engine. I have evaluated the relative merits of the techniques based on their performance and by classifying their limitations and failure cases. One example of the results produced is given in Figure 1.1.

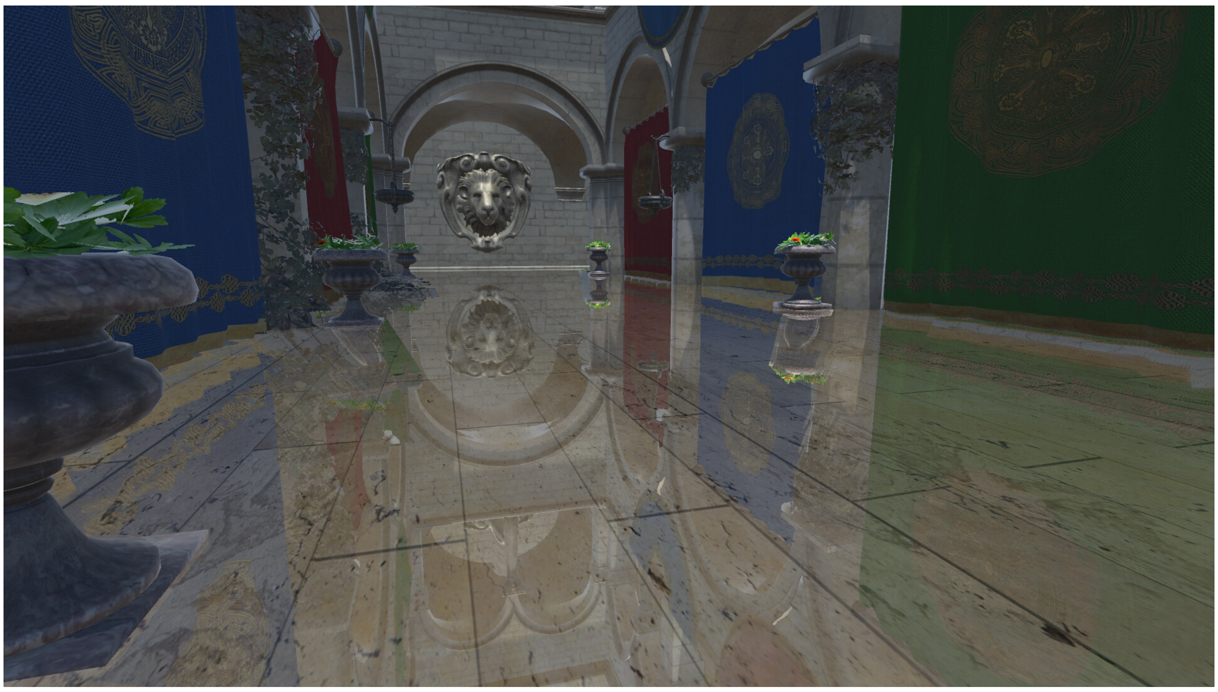


Figure 1.1: The novel reflection technique which I developed, rendering a polished floor in the Unity game engine.

1.1 Motivation

Demand for realistic, real-time 3D graphics is constantly increasing, both in video games and other interactive applications. Real-time, for the purpose of this project, is considered to be a system running with a consistent frame-rate of at least 30 frames per-second (FPS). This is chosen to be sufficiently fast for the human eye to perceive motion, and is an acceptable baseline for performance in video games [1]. This means that graphics must be rendered in less than 33.3 ms, in addition to performing the other tasks inherent in the running of a complex real-time application.

Accurate surface lighting is an important aspect of realistic 3D rendering. Basic approximations for the diffuse and specular components of a surface's colour, such as the Phong reflection model [2], are widely used. However, this model fails to account for the influence of other objects in the scene on the resulting reflection, which is especially notable in "mirror-like" specular reflection. This component is of particular importance in rendering glossy or highly reflective surfaces, both planar and otherwise, such as polished floors, water surfaces or metallic objects. Producing an accurate approximation of this inter-object reflection within such tight time constraints is a challenging problem.

No optimal solution to this problem has been adopted, but a range of different techniques have been utilised in applications previously, which vary significantly in performance and accuracy. More accurate, but more costly, techniques involve re-rendering the scene in some way. Cheaper techniques make use of pre-computation to reduce the run-time costs. This may be done by storing a representation of the scene in an image known as an environment map and using this to render the reflection. The different approaches bring with them different trade-offs in terms of visual artifacts and failure cases, which make them more or less suited to different situations. I expand upon the theory behind each of these techniques in the following chapter.

1.2 Terminology

Below I summarise some domain-specific keywords used in this dissertation, for ease of reference.

- *Texture*: A 2D array of pixels, each pixel typically consisting of four floating-point values (referred to as RGBA). Most commonly used to represent the surface colour of 3D objects (texture mapping) but can be used to store any arbitrary data.
- *Environment map*: An omnidirectional image rendered from a point in a scene - giving a view of the surrounding environment at that location.
- *Cubemap*: A specific implementation of an environment map, mapping the scene onto the faces of a cube (see Section 2.1.5).
- *Normal map*: A texture where each pixel encodes a normal vector.

- *Rendering pipeline*: The steps transforming a 3D scene composed of polygons into a 2D image composed of pixels to display.
- *Shader*: A program running on a GPU, operating as a part of the graphics pipeline. The most common shader types are vertex and fragment shaders. A fragment shader runs once for each pixel or "fragment" of a pixel of a polygon appearing in the output image, taking many inputs and producing an RGBA colour as its output.
- *Draw call*: A function call to the GPU in which a model is rendered using given shaders.
- *Depth buffer*: A texture generated as part of the rendering pipeline in which pixels encode the depth of geometry in the scene (see Section 2.1.6.2).
- *Depth Texture Parallax Corrected cubemaps (DTPC)*: The new technique for reflections I introduce in this dissertation (see Section 3.2.3).
- *OpenGL and Direct3D*: The most common APIs for 3D graphics rendering. The Unity game engine is based on Direct3D, with shaders written in the HLSL language, but can cross-compile to OpenGL platforms.
- *World-space*: A geometric space used to position 3D objects in a scene, with an associated Cartesian coordinate system.
- *Screen-space*: A geometric space in which the camera is at the origin, the xy coordinates map to positions on the screen and the z-axis points down the camera's view direction. This coordinate system maps to the view frustum of the camera.
- *Unity game engine*: A 3D application for developing video games in which the project was built (see Section 2.2.1). Includes an Editor in which most game development occurs.
- *Scene View*: A 3D view within the Unity Editor, used for viewing and modifying scenes.
- *Inspector*: A context-sensitive panel for setting the properties of objects in the Unity Editor.

1.3 Summary

This project focused on comparing and evaluating techniques for rendering reflective surfaces in the context of a real-time 3D application. In the rest of this dissertation, I discuss the research I performed into existing techniques and other relevant areas, as well as the software engineering aspects I considered (Chapter 2). I give details of the process of implementing these techniques in the Unity game engine, along with a novel technique which I developed (Chapter 3). I then evaluate the merits of each of these techniques, along with the success of the project as a whole (Chapter 4).

Chapter 2

Preparation

In this chapter, I present an overview of concepts relevant to the project, based on the research I performed prior to, and during, the implementation stage. I then discuss the aspects of software engineering which I employed to help the project proceed smoothly, along with the requirements which I set for the project.

2.1 Theoretical background

In this section, I begin with a brief introduction to the physical basis for reflections, followed by a range of techniques for rendering reflections. Finally, additional concepts relevant to the new method I implemented are discussed.

2.1.1 Reflections in nature

In classical electromagnetism, visible light is a wave in the electromagnetic field. The path that light takes through an environment may be modelled by considering light in terms of individual rays, which travel in straight lines, a model known as geometrical optics [3, p. 37].

When a ray passes between two media with different refractive indices, it may be reflected. (In general, it may also be refracted or absorbed.) The process follows the laws of reflection: (i) The angle of the incident ray to the normal is equal to the angle of the reflected ray to the normal, (ii) the incident ray, reflected ray and normal are coplanar, and (iii) the reflected and incident rays are on opposite sides of the normal.

Reflection can be classified as diffuse or specular. A reflection is specular, or "mirror-like", if it produces an image, and diffuse if it does not. Very smooth surfaces produce primarily specular reflection, while surfaces with microscopic surface irregularities cause light to be reflected in all directions, giving diffuse reflection.

The proportion of incident light which is reflected by a surface, its reflectance R, can be determined by the Fresnel equations. They give the reflectance for light rays based on their polarisation (s or p), incident angle θ_i, refracted angle θ_t, and the refractive indices, n_1, n_2, of the media [3, p. 106-108].

$$R_s = \left| \frac{n_1 \cos\theta_i - n_2 \cos\theta_t}{n_1 \cos\theta_i + n_2 \cos\theta_t} \right|^2 \qquad R_p = \left| \frac{n_1 \cos\theta_t - n_2 \cos\theta_i}{n_1 \cos\theta_t + n_2 \cos\theta_i} \right|^2$$

In computer graphics, Schlick's approximation [4] is a widely used approximation for the effect of Fresnel reflection.

$$R = R_0 + (1 - R_0)(1 - \cos\theta_i)^5 \qquad \text{where } R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2} \right)^2$$

We can assume that $n_1 = 1$, that is, that one of the media is air. Alternatively, R_0 could be a user-defined constant for a given surface.

Schlick's approximation reproduces the appearance of stronger specular reflections when a surface is viewed at grazing angles.

Geometrical optics is a greatly simplified model of the classical behaviour of light, and furthermore ignores effects arising from the wave-particle nature of light described by quantum electrodynamics [5]. However, as physically accurate simulations of the behaviour of light and matter are far beyond the realm of even the most advanced non-real-time renderers, such a simple model is sufficient for these purposes. The primary concern for real-time rendering is whether the effects are visually convincing to a viewer, regardless of the extent of their physical correctness.

2.1.2 Ray-tracing

Ray-tracing is an approach to rendering based on the ray model of light, which is capable of producing visually accurate reflections [6]. The optimisation made by ray-tracing is that rays are traced backwards from the camera into the scene, approximating the path they would take based on their interactions with scene objects.

For each pixel on screen, a ray is fired into the scene and tested for intersection with each object. The nearest intersection to the camera is found. At the intersection point, further rays can be found, including reflected and refracted rays, and rays towards the light source of the scene. Further ray-traces can be performed recursively using these rays. The colour of the surface at the intersection point, and hence the colour of the pixel, is affected by the influence of each of these rays and the surface properties of the object. The recursive operations allow for physically realistic lighting accounting for other objects in the scene, including mirror-like reflections.

Unfortunately, this approach is computationally expensive as each ray-trace step must perform tests for intersection with some or all of the objects in the scene, and evaluate a

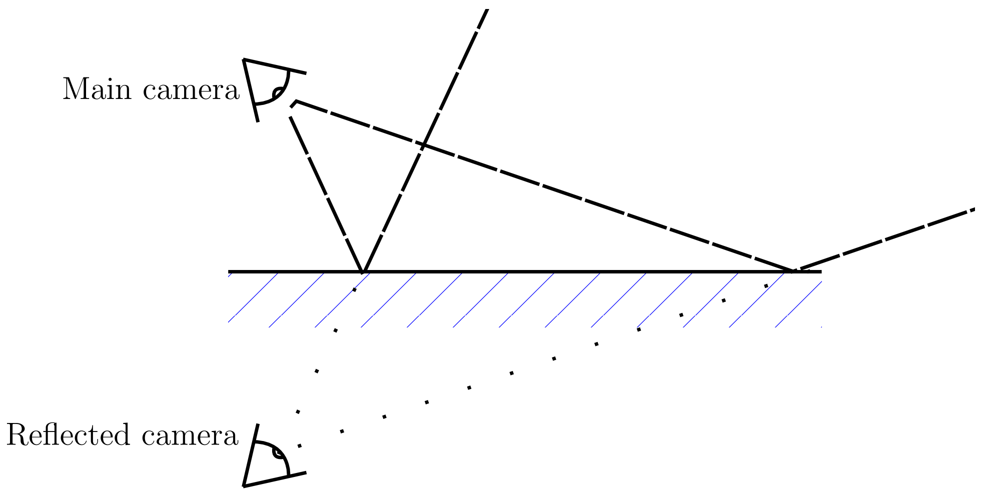


Figure 2.1: A diagram showing the setup for Planar reflections. A secondary camera is placed in the scene, reflected about the plane. The viewing frustums of both cameras are shown as dashed lines.

lighting equation. Additionally there may be many recursive ray-tracing steps per-pixel in order to compute the influence of reflected light. Modern graphics hardware is not well-suited to this type of algorithm, which contains much complex control flow causing stalls on parallelised GPU architectures. GPU architectures have been optimised for the traditional polygon-based graphics pipeline. As a result, the technique is not generally utilised in real-time applications, and faster approximations are needed.

2.1.3 Planar reflections

Within the traditional pipeline, one of the simplest methods for producing reflections is simply to re-render the scene [7].

Another camera is placed into the scene based on the position of the main camera and the reflective surface (Figure 2.1). This camera has a near clipping plane which is aligned with the reflective surface, such that only objects in front of the surface (from the main camera's perspective) are drawn. As a practical example, the technique may be used to render reflections on a flat water surface. The secondary camera would be placed below the water's surface, but the clipping plane means that it would not render any of the scene geometry below the water.

The camera is then rendered separately and composited onto the surface in the main camera image. This achieves results equivalent to those obtained through ray-tracing, but is significantly faster. This is because the secondary camera is rendered in the same way as the main camera, and hence is able to utilise all of the optimisations of the standard rendering pipeline. However, as is apparent from the diagram, the technique is suitable for rendering reflections on planar surfaces only, as for non-planar objects a separate camera would have to be created for each polygon, quickly becoming prohibitively expensive in terms of draw time and memory requirements. This necessitates alternative approaches for the more general case of curved objects.

Note the distinction between this specific Planar reflections technique, and the rendering of reflections on planar surfaces in general, which can be done with any technique. In this dissertation, the capitalised term is used for this technique to avoid ambiguity.

2.1.4 Screen-space reflections (SSR)

As part of the rendering pipeline, depth information about the scene is collected in the form of a depth buffer. The depth buffer is a texture in which each pixel corresponds to the distance to geometry at that point in the scene along the camera's z-axis. Another buffer can contain the normal vectors of surfaces at each pixel. Using this information, combined with the rendered image of the scene, an approximation of ray-tracing can be performed in screen-space (as a post-processing step which occurs at the end of the rendering pipeline). Such a technique can be optimised to run in approximately 1 millisecond on modern game consoles [8]. The limitation is that only objects which appear on-screen can appear in reflections, leaving gaps in the image, or no reflection at all in cases such as a mirrored surface directly facing the camera. Another technique should be employed as a fallback in these cases.

This use of the depth buffer and an approximation of ray-tracing are similar to the alternative approach using cubemaps investigated later in this dissertation.

2.1.5 Cubemaps

An environment map is an omnidirectional image rendered from a point in a scene - giving a view of the surrounding environment at that location. This can be rendered in advance ("offline") and stored in a texture for use during real-time rendering ("online"). This is in contrast to the techniques outlined above, in which no pre-computation is possible.

Environment maps can be used in simulating reflections on curved (or planar) surfaces. Mapping onto a cube is the most widely used technique for storing such an environment map because it simplifies both the process of capturing the images, and the mathematics needed to find a particular direction in the map [9]. The cubemap is captured by rendering the scene six times, once for each face of the cube. Figure 2.2a gives an example of the cubemap texture produced.

The idea behind using cubemaps for reflections is as follows: consider the scene surrounding the reflective surface to be an infinitely distant cube, textured with the cubemap. Then, use the reflected vector from the surface to sample the texture on the surface of the cube, and use this as the reflected colour for the surface. This reduces the entire problem of rendering reflections down to a single texture lookup per pixel.

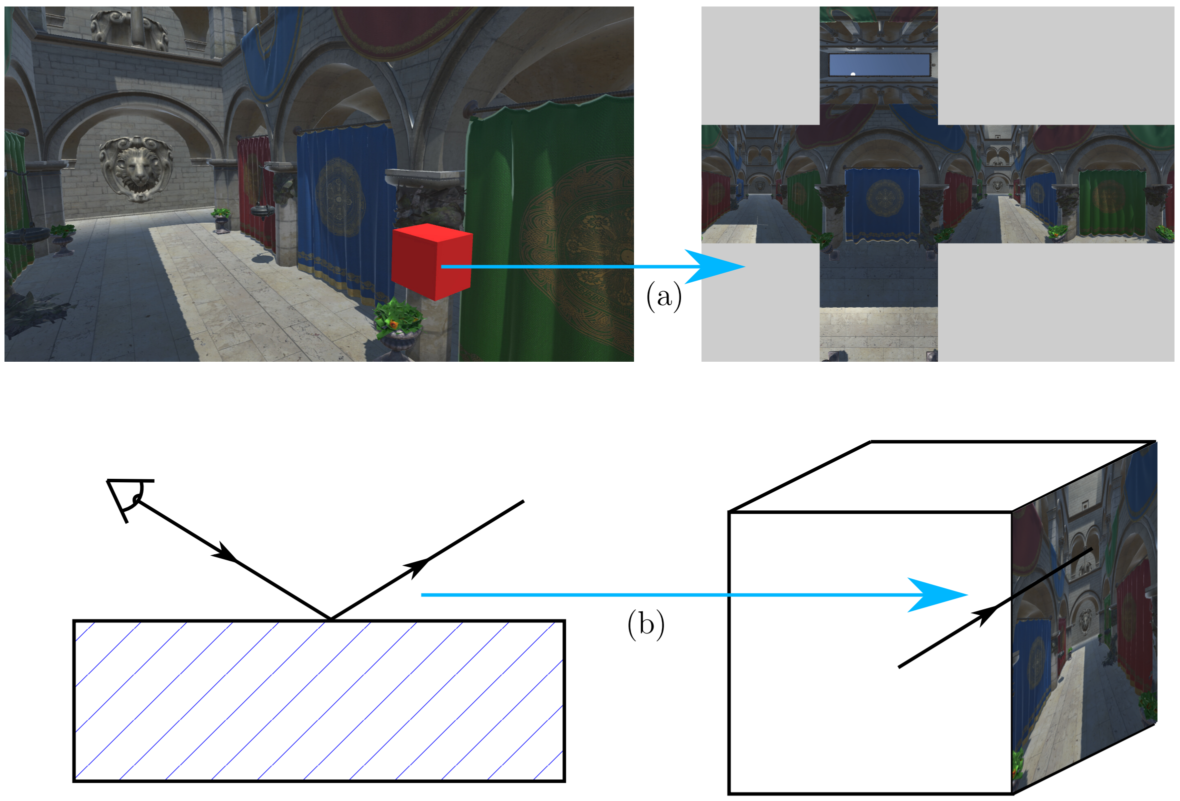


Figure 2.2: A diagram showing how (a) a scene is represented as six faces in a cubemap texture, and (b) how the reflected vector is used to sample the texture.

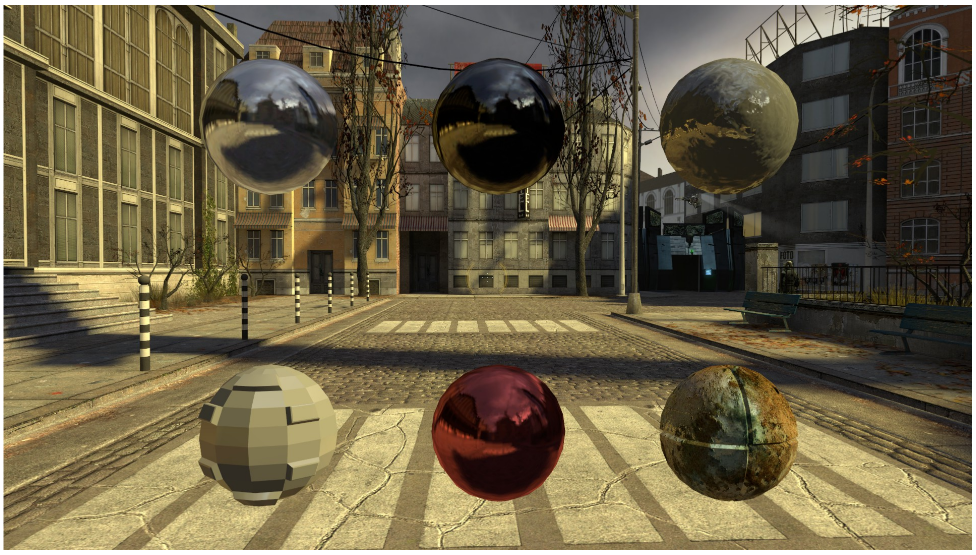


Figure 2.3: Examples of cubemap-based reflections in the *Half-Life 2* engine.

Cubemaps became widely adopted for use in real-time applications following the addition of hardware support for sampling cube textures starting with Nvidia's GeForce 256 GPU [10] in 1999. Both the OpenGL and Direct3D APIs then provided the ability to efficiently sample a cube environment map using a direction vector. Games such as *Half-Life 2* [11] made use of this hardware support to cheaply render reflective surfaces in a realistic environment (Figure 2.3).

However, the technique suffers from a few limitations. Firstly, if the cubemap is captured offline and not recalculated, changes in the scene at runtime are not present in the reflections. Perhaps more noticeably, a cubemap is captured at a single sample position in the scene. As the camera moves away from this point, the reflection becomes increasingly unrepresentative of the scene it is rendered in.

Several techniques can be employed to help alleviate these limitations, whilst maintaining the advantages of using cubemaps. Investigation of these techniques will be the main focus of this dissertation.

2.1.6 Related concepts

The following sections discuss concepts that I learnt about to aid in my implementation of the novel reflection technique.

2.1.6.1 Ray-marching

Ray-marching is an alternative to ray-tracing, which is useful in cases when it is not possible to analytically find the intersection point of a ray with the scene. Instead, the algorithm steps along the ray until it reaches a point which is inside the geometry. This

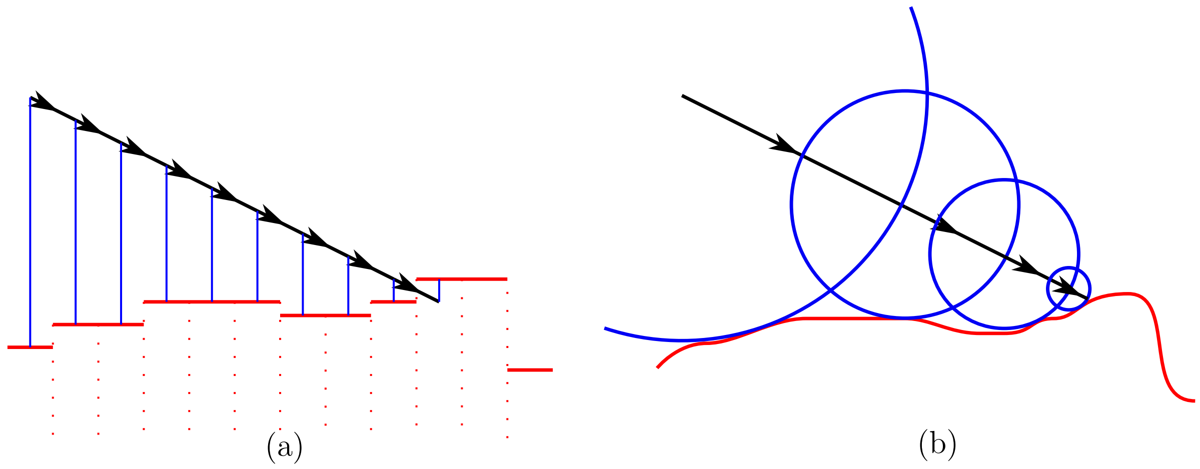


Figure 2.4: (a) A basic ray-march of a heightmap terrain (red). The march proceeds in steps until the ray's current y-position is below the current y-position in the heightmap. (b) A ray-march of an SDF of the same terrain. At each step, the field is sampled and the next step distance is equal to the sampled distance (represented by the blue circle), which is the minimum distance to the surface at the current position.

means that the only required information from the scene is whether a point is inside or outside of the geometry.

This algorithm is useful in rendering heightmaps [12]. A heightmap is a 2D texture in which each pixel encodes the height of the scene at that point. The ray has intersected the scene when the its y-coordinate is less than the height at the current position in the heightmap, as shown in Figure 2.4a.

The algorithm can be slow as many steps may be needed in order to get a sufficiently accurate intersection and to avoid stepping through small objects. Several techniques exist to speed up ray-marching. One such method is signed distance field (SDF) ray-marching [13]. In a signed distance field, the scene is defined as a function giving the distance to the surface of the geometry at every point. The ray-marching algorithm can then be adapted to sample the SDF at each step. The length of the next step taken can be equal to the sampled distance, because the scene geometry must be at least this far away along the ray direction. Figure 2.4b gives a representation of this process in two dimensions, where circles represent the values obtained from the distance field at each point.

I investigated the use of a ray-marching algorithm in the novel cubemap reflection method, which I discuss in the Implementation chapter.

2.1.6.2 The depth buffer

In the GPU rendering pipeline, as part of the rendering process a texture called the depth buffer is filled. Each pixel in the depth buffer contains the distance along the camera's

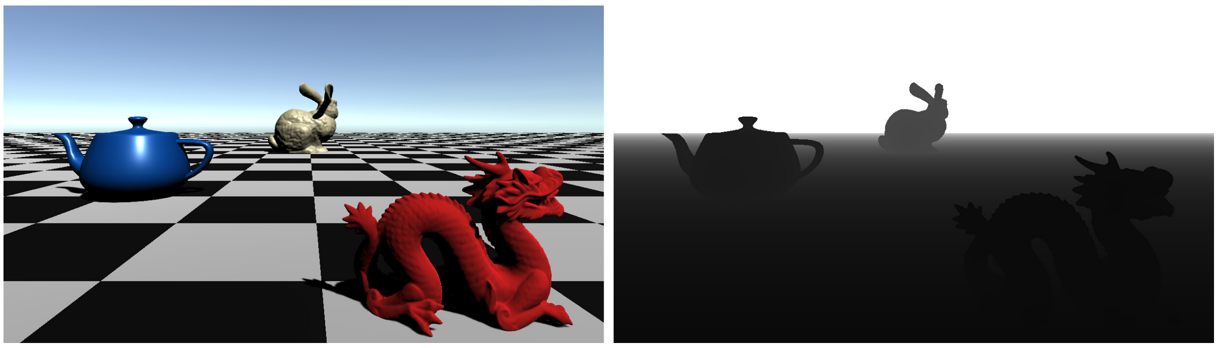


Figure 2.5: A rendered scene and the contents of its depth buffer.

z-axis of the geometry at that point in the scene. It is used when rasterising triangles to ensure that pixels are only filled when the triangle being rendered is in front of the geometry already rendered at that pixel. The information stored in the depth buffer can be extracted by shaders, for use in post-processing effects such as screen-space reflections. The format in which depth buffer values are stored varies between platforms and is non-linear. This must be accounted for when writing shaders which use the depth buffer.

Using render-to-texture, the depth information can also be stored to the disk as an image for future use. As a result, a camera can be rendered and its depth buffer used to generate a heightmap representation of the scene. Furthermore, it is possible to create an omnidirectional depth map by extracting the depth buffer in each direction when capturing cubemaps. I discuss the steps needed to perform this operation and the use of the generated depth cubemap in the Implementation chapter.

2.2 Software engineering techniques

2.2.1 Choice of tools

I chose to use the Unity game engine[1] for my project. The reasons for this decision were twofold. Firstly, it allowed me to focus my limited time on implementing components directly related to the aims of my project, as boilerplate components such as cameras, scene editing and setting up a rendering context are handled by the engine. Secondly, Unity is an engine widely used in commercial projects and hence the results achieved by my implementation should be reflective of what is possible in real-world applications of the techniques.

I used Microsoft Visual Studio as my IDE because of its integration with Unity.

For the purposes of evaluation, I wrote additional Unity scripts and made use of the engine's in-built profiler. For analysing GPU performance, I used RenderDoc[2], an appli-

[1]`https://unity3d.com/`

[2]`https://github.com/baldurk/renderdoc`

cation which can be used to capture frames rendered by Direct3D programs, and replay them later, allowing me to make repeated timings of individual draw calls.

Finally, I used Git for version control. The project is mirrored on a private GitHub repository as well as two personal machines. Branches were used to work on experimental changes while maintaining a functioning build.

2.2.2 Project structure

The project consists of a set of independent components; the different techniques being investigated. The implementation of these was kept separate, however going into the coding stage it was important to maintain a consistent structure for each, for ease of use and to allow them to be easily interchanged during the evaluation stage. This would be somewhat challenging due to the different combinations of GPU and CPU code needed for each technique, but was achieved by defining an abstract technique superclass which encapsulates basic properties. Each implemented technique extends from this, performing the necessary setup and runtime behaviour for that technique.

Other components included methods for capturing cubemaps and recovering information from the depth buffer. Each was required to function independently of other parts of the project and to integrate closely with the Unity Editor. For example, each of the techniques needed a customised graphical user interface (GUI) within the Editor.

2.2.3 Methodology

Overall, I chose to use an iterative approach to designing the project. In reimplementing existing techniques, the more complex techniques tended to build upon the approaches used by the basic ones, so it made sense to schedule work on them sequentially, as the knowledge gained could be transferred between each step of the process. In implementing the novel techniques, the process was more exploratory, so I would start with the most naïve implementation and then iterate upon this, introducing optimisations and improvements.

Finally, when there was a dependence between different parts of the project, such as cubemap capture and the cubemap reflection techniques, I would use placeholder inputs, such as example cubemap images sourced online, or manually created depth textures, to allow implementation to proceed unhindered.

2.3 Requirements analysis

Before starting the implementation, I defined the requirements of my project based upon the Success Criteria and extensions which outlined in the Project Proposal (Appendix B).

I then used the MoSCoW prioritisation technique to organise these goals for implementation. This technique defines four levels of priority, *Must have*, *Should have*, *Could have* and *Won't have, but would like.*

The requirements which I decided on for the project are outlined below.

1. Implementation of the basic cubemap rendering technique, along with at least one pre-existing method for rendering cubemaps with parallax correction.

   This is the most basic goal of the project, to investigate these techniques for the purposes outlined above. **Must have.**

2. Implementation of the proposed depth-based parallax correction technique, along with a method for capturing cubemaps with the required depth information.

   This is the main project goal. As it is a new technique there may be more challenges encountered, but an implementation is a crucial part of the project. **Must have.**

3. Implementation of each technique with a uniform interface and integration with the Unity Editor.

   This is important for usability and performing the evaluation efficiently. **Should have.**

4. Implementation of alternative techniques, including ray-tracing, Planar reflections and screen-space reflections.

   This is important to gain a good understanding of the relative merits of the techniques being investigated. **Should have.**

5. Each method (excluding ray-tracing) should be optimised to perform sufficiently well for use in a real-time application, for example, running at 30 FPS or higher on a typical gaming computer.

   This is not a strict performance requirement and any implementation should be able to reach it. **Must have.**

6. Completion of an evaluation based on: i) Comparison of the relative performance of each technique, with and without optimisations applied. ii) Comparison of the visual results, with one or more alternative techniques for real-time reflections, and with an accurate reference image. iii) Investigation and classification of the failure cases for each technique (for example, cases in which the technique incurs a much greater performance cost, or produces a particular visual artifact when compared to a reference).

   The evaluation is a crucial part of the project. **Must have.**

7. Extension: Additional reflection features, such as bump mapping, masking and blending with a surface texture.

   An extra feature that should be easy to implement but will produce results more representative to those used in an actual system. **Could have.**

8. Extension: Cubemap blending - to investigate methods for interpolating smoothly and efficiently between cubemaps as the camera moves between areas.

   An extra feature that could be implemented if extra time is available but is not key to the goals of the project. **Could have.**

9. Extension: Dynamically rendered cubemaps and integration of dynamic objects into static cubemap reflections.

   I decided that these tasks would be too time consuming within the project schedule that was available and so were not prioritised. **Won't have.**

This set of requirements allowed me to focus my implementation time on the highest priority aspects of the project.

2.4 Starting point

I had some pre-existing knowledge writing shaders in HLSL (High Level Shading Language) as a result of a summer internship at a video games company, Studio Gobo. The suggestion for this project originated with a programmer from Studio Gobo, Huw Bowles, who provided references for my initial research and suggested investigating the depth-based approach using ray-marching.

The project was built using the Unity game engine. As a result, all of Unity's features were available for me to use. Most notably these include a full rendering system, the GameObject system (in which objects in the scene can be composed from a GameObject and a list of scripts known as Components), and a graphical interface for scene editing. No Unity-provided reflection techniques were used by in my project. My starting point was an empty Unity project.

Chapter 3

Implementation

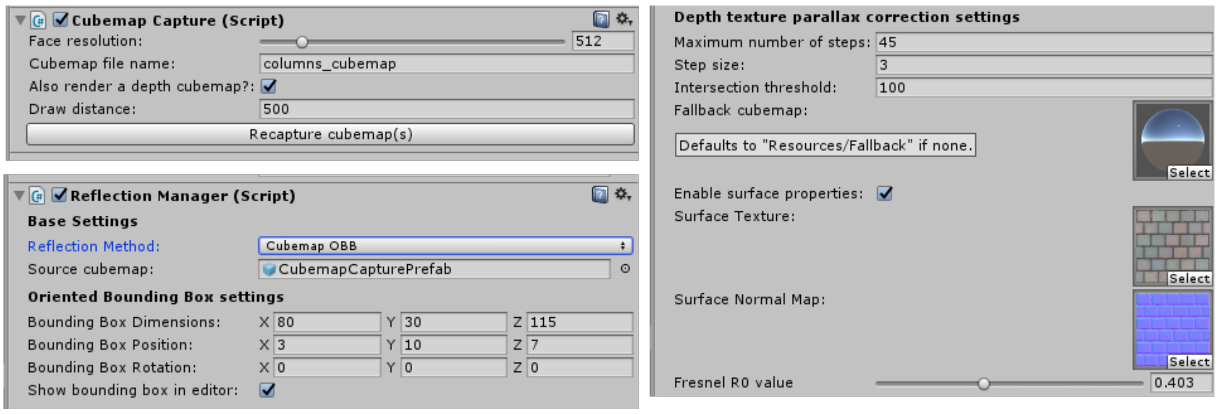
In this chapter I give details of how I implemented a range of reflection techniques in the Unity game engine. I give examples of the combination of CPU and GPU code required for each technique and the challenges they presented. I go on to give a detailed explanation of the process of implementing the novel reflection technique, and the challenges that were encountered.

3.1 Project setup

The project takes the form of a Unity engine project directory, in which source files are divided into subdirectories by type and purpose. For ease of use, I designed the project to integrate tightly with the Unity Editor. This had the added benefit of making it easy to write scripts to automate the testing of different reflection techniques during the Evaluation process.

The main user-facing components of the system are outlined below:

- *CubemapCapturePrefab*: A prefab is a preconstructed GameObject that can be placed into any Unity scene. This prefab has a script attached which is responsible for rendering and storing cubemaps and depth cubemaps for use in the rendering techniques which I implemented. The script is associated with a custom user interface in the Unity Inspector to allow for easy tweaking of parameters, as seen in Figure 3.1.
- *ReflectionManager*: This class can be attached as a Component to any GameObject in order to render a reflective surface on that object. The ReflectionManager object contains a list of references to Helper objects for each reflection technique. These Helper objects perform the setup and runtime behaviour of each technique, including passing parameters to the GPU. Each Helper is constructed only when its technique is selected. The script has a custom interface in the Inspector providing a drop-down menu for switching between techniques and displays controls specific to the currently



Cubemap Capture (Script)
Face resolution: 512
Cubemap file name: columns_cubemap
Also render a depth cubemap?: ✓
Draw distance: 500
Recapture cubemap(s)

Reflection Manager (Script)
Base Settings
Reflection Method: Cubemap OBB
Source cubemap: CubemapCapturePrefab
Oriented Bounding Box settings
Bounding Box Dimensions: X 80 Y 30 Z 115
Bounding Box Position: X 3 Y 10 Z 7
Bounding Box Rotation: X 0 Y 0 Z 0
Show bounding box in editor: ✓

Depth texture parallax correction settings
Maximum number of steps: 45
Step size: 3
Intersection threshold: 100
Fallback cubemap:
Defaults to "Resources/Fallback" if none.
Enable surface properties: ✓
Surface Texture:
Surface Normal Map:
Fresnel R0 value 0.403

Figure 3.1: An example of the custom GUIs and tooltips for *CubemapCapture* and *ReflectionManager* which appear in the Unity Editor. The latter is context-sensitive depending on the technique being edited (shown lower-left and right).

active technique. It also draws additional information, such as bounding boxes, into the 3D Scene View of the Editor.

The techniques can all be previewed in the Editor as well as at run-time. They can also be changed at run-time, either by a method call from another script or using keyboard controls.

The project takes advantage of inheritance and interfaces for the purposes of polymorphism and code reuse. Helper classes extend from a common abstract class allowing them to be used interchangeably throughout the project, despite their very different implementations across the GPU and CPU.

3.1.1 Capturing cubemaps

The ability to create new cubemap textures is implemented in the *CubemapCapture* script. When attached to a game object (eg. in the *CubemapCapturePrefab*) placed somewhere in the Scene, it can capture cubemaps at that point.

The basic procedure for creating a cubemap is as follows. For a given `FaceResolution` an output texture of size `(4 * FaceResolution) x (3 * FaceResolution)` is created. This will contain the final cube net which is written to disk. A render texture the size of one face is also created. A render texture is a texture on the GPU which a camera can render to. A camera with a 90° field-of-view is created at the capture point, and is rotated to each of the six axis directions. The scene is rendered to the render texture each time, the resulting image is copied onto the CPU output texture in the correct position in the net. Finally, the output texture is saved to disk.

3.1.1.1 Capturing depth

As discussed in Section 2.1.6, a shader can be used to output the depth texture that is created in rendering a scene. Therefore we can save the depth map of a scene to disk by rendering the scene, with such a shader applied, to a render texture and then reading and storing the resulting texture.

A few extra steps are required to obtain usable values from the depth buffer. Firstly, the hardware depth buffer is stored in a non-linear format. This format varies between platforms, but Unity provides a cross-platform shader macro to convert from the hardware value to a normalized floating point value. Then, a means of storing this value in a texture without loss of precision is needed. In theory, a floating point texture format should make this trivial, but in practice it was found that Unity's support for floating point textures was limited and precision was lost. Instead, I wrote a function to discretise the value and store it in the RGBA channels of a standard texture, from most significant to least significant bits.

To decode the texture back into an actual world-space distance in another shader, the inverse process is needed. In addition, the depth value obtained needs to be denormalized. This is done by multiplying the value from the texture by the distance of the far clip plane of the original capture camera. In my implementation, this value can be passed from the *CubemapCapture* script, along with other useful information such as the world-space location of the sample point.

There is one further problem to consider. The depth value contained in the buffer is the distance to a point along the camera's z-axis. In the case of heightmap rendering, this is the correct value that is needed. However, the value that will actually be needed when rendering the depth cubemap is the straight-line distance from the sample point to the position, along the direction vector being sampled. See Figure 3.2 for a visual explanation. I performed this additional correction step when rendering the cubemap face by using the texture coordinate of the pixel to calculate the straight-line distance.

When the resulting texture file is imported back into Unity, it must be set up to use pointwise (unfiltered) sampling, and also must be flagged as a "Truecolor" texture to prevent Unity applying lossy compression and destroying the depth information. The latter point caused many weeks of difficulty as it was not easy to discover what was causing the problem.

3.2 Cubemap-based techniques

I will now go on to discuss the implementation of each of the reflection techniques investigated. I begin with the cubemap-based techniques which were my primary focus.

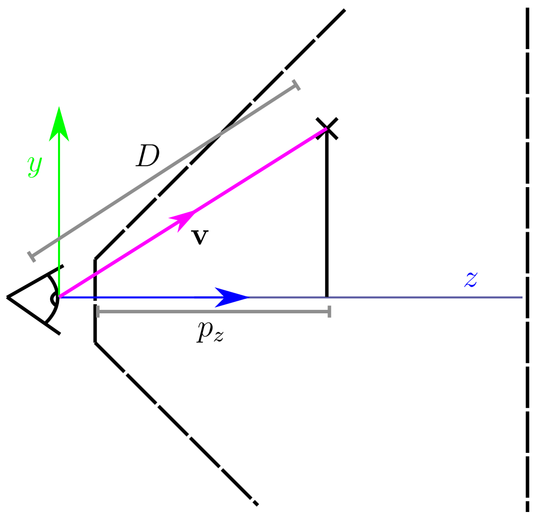


Figure 3.2: A diagram showing the capture of one cubemap face. When sampled in a direction $\hat{\mathbf{v}}$, the resulting distance will be the axis-aligned distance along the z-axis of the camera that captured the face, $\mathbf{p}_z$. What is actually needed is the straight-line distance, D, to the scene geometry in the direction $\hat{\mathbf{v}}$. Given knowledge of the near-clip distance and the texture coordinates of the pixel in question, it is trivial to find D using similar triangles. This correction is performed when the depth buffer is recovered, before saving into the cubemap texture.

3.2.1 Uncorrected

The most basic method of producing a reflection given a cubemap image is to calculate the reflection vector from the object's surface and use this to sample the texture directly. Only a basic fragment shader[1] is needed, which requires access to the surface normal and the view vector (that is, the direction of the camera from the surface).

The basic fragment shader can be extended to create more visually interesting effects. A normal map can be used to define the normal vectors used in the reflection calculation, simulating surface "bumpiness" at the sub-polygon scale. A base texture can be added to define the surface's colour, and blended with the reflection colour. The weighting of reflection colour versus surface colour can be a constant, or dependent upon the reflected angle, as in Fresnel reflection (see Section 2.1.1). Each of these effects is shown implemented in Figure 3.4 and can be easily used with any of the other cubemap techniques investigated. Texture and normal mapping are omitted from subsequent screenshots to make the reflected image itself easier to see.

As mentioned in Section 2.1.5, this technique is subject to parallax error because it makes the assumption that all geometry appearing in the cubemap image is an infinite distance away from the sample location. The following techniques aim to reduce the effects of parallax error in the image produced.

[1]Typically known as a pixel shader in the context of HLSL.

```
float3 incidentRay = −input.viewDirection;
float3 n = input.surfaceNormal;
return texCUBE( CubemapTexture, reflect( incidentRay, n ) );
```

Figure 3.3: Example of the contents of a fragment shader for basic cubemap reflections. `reflect` is an intrinsic HLSL function producing a reflection vector given an incident vector and surface normal. `texCUBE` samples a cube texture.

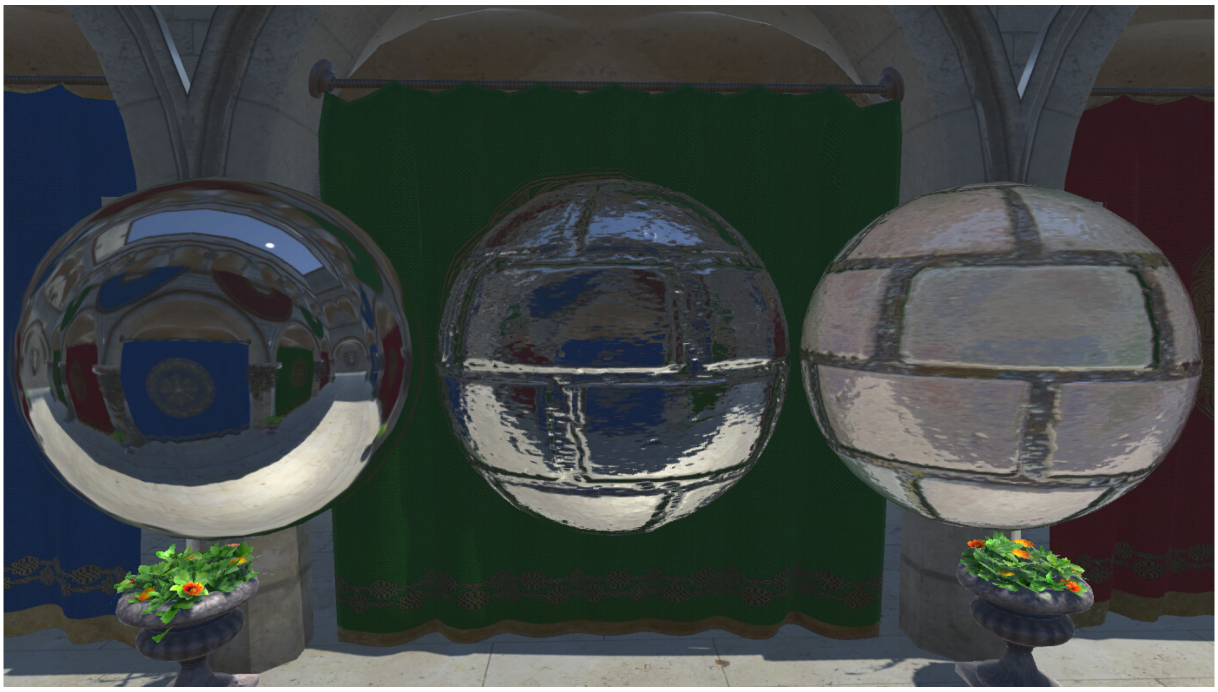


Figure 3.4: From left to right: 1) Basic cubemap reflections. 2) Reflections with the surface's normal vectors determined by a normal map. 3) Reflections with a normal map using Schlick's approximation of Fresnel reflection to determine the influence of surface colour versus reflected colour.

3.2.2 Proxy geometry parallax correction

To correct for the parallax error in cubemap reflections, two things are needed; the position at which the cubemap was captured in the scene (the sample point), and the positions of geometry in the scene. The first is readily available (simply read the position of the *CubemapCapture* object) but the latter is more difficult. It is needed to find the position at which the reflected ray intersects with the scene, so that the correct location in the cubemap can be sampled. The brute-force approach is ray-tracing the scene, which is too costly and defeats the point of using a cubemap.

This technique instead defines a simpler proxy geometry to represent the scene [14]. This geometry is a 3D primitive chosen such that it is cheap to calculate ray intersections with. If we can find the position at which a reflected ray hits this approximation of the scene, and we know the cubemap sample point, we can find the direction of this intersection point in the cubemap.

The reflection produced is then accurate given the assumption that the actual scene is close to the proxy geometry.

The per-pixel algorithm is the following:

1. Find the reflected ray from this point on the surface.
2. Find the closest intersection point of the ray with the proxy geometry, in world-space.
3. Given the location of the intersection, and the sample location of the cubemap, find the direction of the intersection point from the cubemap.
4. Use the direction vector to sample the cubemap texture.

A diagrammatic form of this algorithm is shown in Figure 3.5.

I chose to first implement this technique using an axis-aligned bounding box (AABB) as the proxy geometry. An AABB can be entirely defined by its maximum and minimum coordinates.

There are six axis-aligned planes with which to test for intersection, giving six equations:

$$\begin{aligned}
\mathbf{u}_x + t\hat{\mathbf{v}}_x &= \mathbf{m}_x & \mathbf{u}_x + t\hat{\mathbf{v}}_x &= \mathbf{n}_x \\
\mathbf{u}_y + t\hat{\mathbf{v}}_y &= \mathbf{m}_y & \mathbf{u}_y + t\hat{\mathbf{v}}_y &= \mathbf{n}_y \\
\mathbf{u}_z + t\hat{\mathbf{v}}_z &= \mathbf{m}_z & \mathbf{u}_z + t\hat{\mathbf{v}}_z &= \mathbf{n}_z
\end{aligned}$$

where $\mathbf{u}$ is the world-space position of reflection, $\hat{\mathbf{v}}$ is the reflected vector, and $\mathbf{m}$ and $\mathbf{n}$ are the minimum and maximum bounding box coordinates. The smallest positive value for t is the intersection distance with the AABB. We can treat these as two vector equations in order to take advantage of GPU vector operations and rearrange to find two vectors of t, from which we take the smallest positive value.

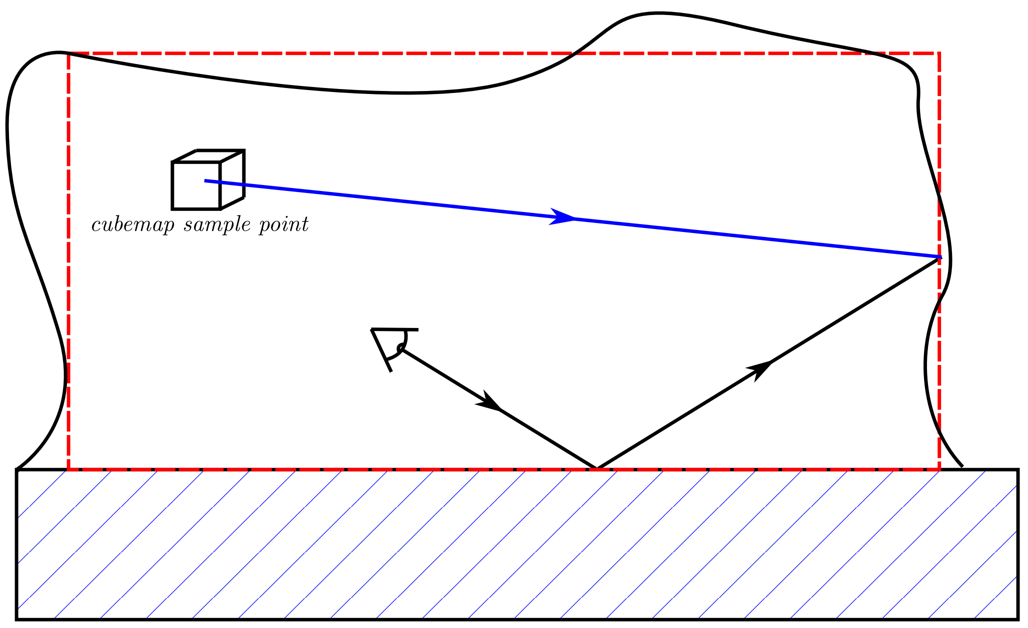


Figure 3.5: A diagram showing how the intersection point of the reflected ray with a proxy geometry (red) can be used to get a corrected direction vector (blue) to sample the cubemap texture.

This algorithm is performed within the fragment shader, and as a result there is no run-time CPU overhead. The CPU simply sets up the shader and passes in the min. and max. bounding box coordinates given a user-defined position and scale. The bounding box itself rendered within the Editor Scene View to help the user define a bounding box close to the actual scene geometry. An example of the reflections produced using an AABB proxy geometry are shown in Figure 3.6.

I was able to remove the restriction of axis-alignment without increasing the complexity of finding the intersection point. To do this, I defined a rotation matrix on the CPU based on a user-specified rotation, which was passed into the shader. This transformation is then applied to $\mathbf{u}$ and $\hat{\mathbf{v}}$ before finding the intersection. The intersection distance is then used to find the intersection point in "un-rotated" world-space coordinates. Therefore, the additional cost of using an oriented bounding box (OBB) is two matrix-vector multiplications per pixel. An OBB will be preferable to an AABB in cases where the scene is close to a box in shape but not aligned with the coordinate axes, for example a sloping corridor.

Other types of proxy geometry are possible, as long as their intersection with a ray can be cheaply computed in a shader.

I also implemented sphere proxy geometry. Taking the vector equation for a sphere,

$$|\mathbf{x} - \mathbf{c}|^2 = r^2,$$

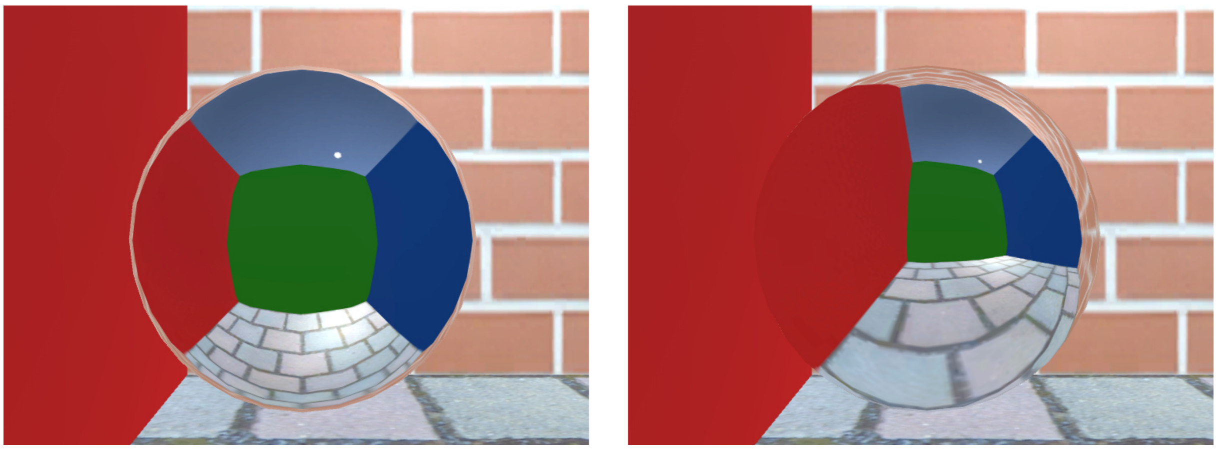


Figure 3.6: Cubemap reflections on a spherical surface. The cubemap was captured at the centre of the room and the surface placed in one corner. Left is the uncorrected reflection. Right is the reflection with AABB parallax correction applied.

and substituting in the equation for a ray,

$$\mathbf{x} = \mathbf{u} + t\hat{\mathbf{v}},$$

produces a quadratic equation in t. In the spherical proxy parallax correction shader, this quadratic is solved and $\max(t_1, t_2)$ is taken as the intersection distance.

3.2.3 Depth texture parallax correction (DTPC)

The catalyst for investigating this technique was the idea that, instead of using an approximate, and user-defined, proxy geometry for the parallax correction algorithm, information about the actual scene's geometry could be used instead. This information could be obtained from a depth map captured at the cubemap sample point, and stored in another cubemap texture (the implementation of which was explained in Section 3.1.1). This "depth cubemap" effectively defines the surface of the scene surrounding the cubemap.

It is not possible to find analytically the intersection point of a ray with this surface. Instead, a ray-marching-based approach must be used to iterate towards intersection.

Due to the unfamiliar nature of this technique, and the difficulties inherent in debugging shader code, in implementing this technique I worked towards the final goal in stages.

3.2.3.1 Heightmap rendering

I began by implementing a method of heightmap rendering using ray-marching. A heightmap of an existing Unity scene could be captured using the process in Section 3.1.1, allowing for easy comparison of the scene rendered using the traditional pipeline

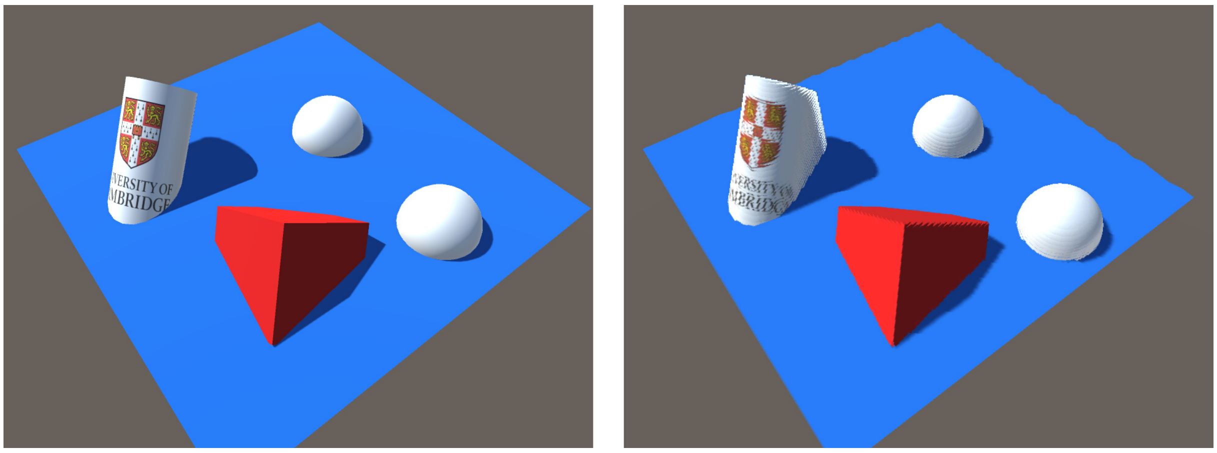


Figure 3.7: A simple scene in Unity. Left: rendered tradtionally. Right: rendered by ray-marching a 512×512 pixel heightmap. Note how parts of the scene which are not visible from above cannot be rendered accurately.

and the ray-marching method. The algorithm within the heightmap rendering shader proceeds as follows:

1. Fire a ray from the camera position into the scene.
2. Step along the ray in steps of a fixed size.
3. At each step, convert from world-space coordinates to texture coordinates to sample the heightmap.
4. If the sampled height value is greater than the current height of the ray, stop and return the colour of the image texture at the current texture coordinates.

An example of the results of this algorithm can be seen in Figure 3.7. The performance of the algorithm and the quality of the results are directly dependent on the maximum number of steps, and step size, used in the shader.

3.2.3.2 Depth cubemap rendering

The main change from rendering a heightmap, is from sampling distance from a 2D texture using texture coordinates, to sampling distance from a 3D surface using a direction vector. The new algorithm then becomes the following:

1. Fire a ray from the position of reflection along the reflected direction $\hat{\mathbf{v}}$.
2. Step along the ray in steps of a fixed size.
3. At each step, find the direction of the current position from the cubemap sample point and sample that direction in the cubemap texture.
4. Find the difference, ε, between the distance sampled from the texture, with the actual distance between the sample point and the current position. If $|\varepsilon|$ is less

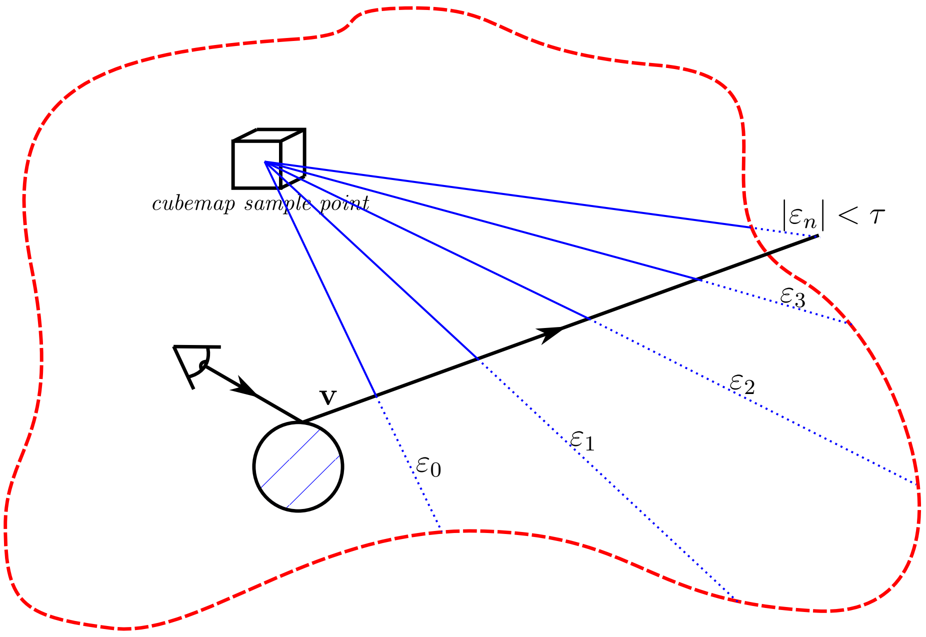


Figure 3.8: A diagram giving a simplified representation of the DTPC algorithm. Red dashes represent the depth information stored in the captured cubemap. The ray-march proceeds in steps in the direction of the reflected ray $\hat{\mathbf{v}}$. At each step a depth sample is taken (blue) and the distance, ε, from the current ray position to the sampled position is found. The ray-march terminates when either the maximum number of steps is reached or ε meets the stopping criteria.

than a given threshold τ, stop the ray-march and sample the image cubemap in the current sample direction.

A diagrammatic form of the algorithm is given in Figure 3.8. An example of the results of the basic algorithm is shown in Figure 3.9.

The algorithm above makes the assumption that *the distance of the ray from the surface in the direction from the cubemap* (ε) is similar to *the distance of the ray from the surface along the direction of the ray*. There are some obvious cases when this would not be a correct assumption, for example the ray approaching a surface close to the normal, when the sample point is close to perpendicular to the ray direction. However the error introduced in such cases is not significant when the step size is relatively small, and should be corrected in the next step if an overstepping is caused.

It may be preferable to change the stopping criteria, from $|\varepsilon| < \tau$, to $-\tau < \varepsilon < 0$. This way reflected objects will not appear to protrude further forward than the original object, and increasing the threshold makes the reflected objects appear thicker in the occluded direction.

Because of the fixed step size, a quantisation error is visible on large surfaces as the intersection distance changes in number of steps. I used a simple technique to reduce the

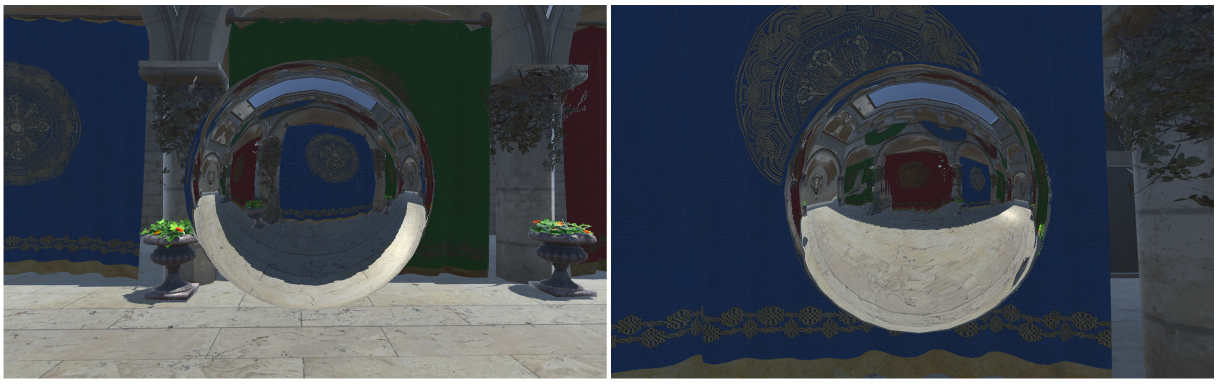


Figure 3.9: Reflections rendered on spherical surfaces in different areas of a scene, using the DTPC technique. The same source cubemap is used, but the result has been parallax corrected so that the correct reflected image appears despite displacement from the sample point. The right hand image shows some evidence of artifacts due to the occlusion failure case.

visibility of this effect. When the distance is detected to be within the threshold, the most recent value of ε is added to the ray distance before sampling the image. This reduces the visibility of quantisation in many cases, again assuming a relatively small step size.

3.2.3.3 Optimisations

I was able to implement the technique as outlined above to be fast enough to run in real-time. However, I wished to find further optimisations to make it more competitive with other techniques and give a more fair performance comparison.

The goal in any optimisation would be to reduce the number of ray-march steps that are required, as each step requires an additional texture sample. Simply increasing the step distance and reducing the maximum number of steps will improve performance but will lead to objects in the scene being stepped over, and more visually obvious quantisation effects. The extent to which each of these effects is visible will depend on the scene, and also on the reflective surface, as curved or bumpy surfaces will hide some imperfections. These issues are discussed further in the Evaluation.

Another approach is needed to further improve performance without compromising the visual results. The first attempt I took was to investigate an approach similar to distance field ray-marching. Instead of having a distance field sample to determine the distance to the surface, I decided to approximate it by $k\varepsilon$, for some $k \in (0, 1]$. This should allow much larger steps to be taken towards intersection. However, I found that in implementing this method the algorithm would converge to the incorrect distance, or fail to converge at all, in many cases. The results obtained also depended greatly on the size of the initial step that was taken. While good-looking and fast results were achievable in some restricted cases, I concluded that this approximation was not valid in general.

Given that the reason for failure in this algorithm is due to skipping over objects, it would be useful to have more information available at each step about potential intersections. One way to do this is to introduce a data structure known as mipmaps. Mipmaps are a chain of successively lower resolution versions of a texture. Typically, the lower level versions are calculated by a process such as bilinear interpolation. Instead, I could change the sampling method to take the minimum value from the local pixels at the higher mipmap level. This gives a data structure in which sampling the lower levels provides successively more conservative estimates of the distance to the surface. The algorithm can then be altered as such: starting at the lowest mipmap level, take large step sizes until intersection is found ($|\varepsilon|$ is less than the threshold), then increase the mipmap level until the current position is no longer an intersection. Then continue the march at the higher level, and with a corresponding smaller step size. The ray-march terminates when an intersection is found at the highest mipmap level.

It transpires that this approach has been used previously in the context of rendering heightmaps [15]. I believe it has the potential to give a significant speed-up to the algorithm that could make it more suitable for use in the industry. However, given the time limitations of my project schedule, I have not completed implementation of the approach and have left it as an area for future work.

3.2.3.4 Failure cases

The primary failure case for this technique, and for any technique using parallax corrected cubemaps, is the case in which objects should appear in the reflection which are occluded from the perspective of the sample point. This means there is no information in the cubemap about what should appear in these parts of the reflected scene.

I considered three possible approaches to handle the occlusion case.

- **Stop the ray-march as soon as the ray enters an occluded part of the scene**. This corresponds to setting the stopping criteria as simply $\varepsilon < 0$. It is faster as it stops the march early but results in objects having unnatural-looking "tails" which obscure the occluded areas.
- **Continue the march**. Either the ray re-enters a non-occluded part of the scene or continues to infinity. In the second case this can optionally be detected (because the number of steps is equal to the maximum) and an alternative Fallback cubemap image can be used.
- **Start a second ray-march**. A second set of cubemaps can be captured to contain occluded parts of the scene, for example by culling foreground objects from the scene or setting a larger near clip distance when capturing. Then, when a ray is found to go to infinity, another ray-march using the "distant" cubemaps can be started to render this pixel.

An implementation of the third approach was quite straightforward but given the cost of doing an entire second ray-march added to the overhead of additional branching in the shader, I decided this was not a worthwhile extension. Instead, the choice can be made from the first two approaches and may be dependent on the appearance of the scene itself.

3.3 Other reflection techniques

I now go on to discuss my implementation of other techniques for real-time reflections, which will form the basis of my comparisons with the cubemap techniques in the Evaluation.

3.3.1 Planar reflections

Accurate reflections on planar surfaces are achieved by re-rendering the scene from another viewpoint. In my implementation, the *PlanarHelper* obtains a reference to the currently active camera, along with the position and normal of the plane. This is used to create a second camera, the transformation matrix of which is modified by reflection about the reflective plane.

An oblique projection matrix is calculated [16] for the secondary camera, such that the near clip plane of the viewing frustum is modified to match the reflective plane (meaning that the near clip plane is no longer perpendicular to the camera's z-axis) [17]. This is a cheap way to clip all objects which are below the reflected plane in the scene. Not doing so would cause artifacts, as objects that are behind the surface could appear in the reflection.

The secondary camera renders to a texture, which is passed into a shader applied to the plane itself, so that the reflection image appears on the plane in the main camera's image. An example of the results is shown in Figure 3.10.

One advantage of this technique is that recursive reflections are possible. As the secondary camera re-renders the scene, it can also render other planar reflective surfaces that would be visible in the reflection. To prevent an infinite recursion depth, a global counter of the current depth of rendering is maintained so that rendering stops at a user-defined depth.

3.3.1.1 Optimisation

Because the technique renders to a texture, the resolution of the reflected image can be scaled independently of the screen resolution. Additionally, harsher criteria can be applied to what can be rendered in the reflected scene. A nearer far clipping distance can be defined. A culling mask can define objects from the scene not to appear in the reflection, for example to prevent small but high-polygon objects being re-rendered. Each of these optimisations can be applied using the custom Inspector controls I implemented.

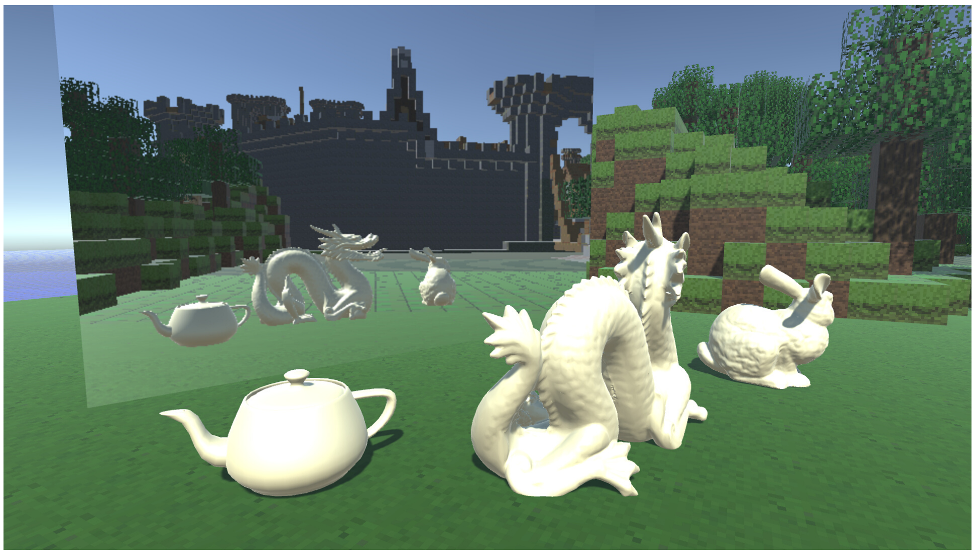


Figure 3.10: A reflective plane rendered with Planar reflections. Unlike in cubemap-based techniques, moving objects can appear in the reflection. The reflected scene is rendered at half the screen resolution.

3.3.2 Ray-tracing

I chose to implement ray-tracing as a means to obtain an "accurate" reflection image, as a point of comparison with the other techniques, rather than as something appropriate for a real-time system. The ray-tracing algorithm can be implemented on either the CPU or the GPU. A GPU ray-tracer would give significantly better performance because of the parallelism available. However, for the purposes of this project I chose to implement the technique on the CPU. I did this because the algorithm must integrate with Unity, to be able to calculate ray intersections with arbitrary objects in the Unity scene and access their properties such as colour, texture and reflectance. Doing this on the GPU presents many challenges and was a task far beyond the scope of this project.

The CPU algorithm I implemented was able to make use of Unity's ray-casting system which is designed for uses such as physics collisions. It allowed me to obtain the intersection point of any ray and a reference to the object it intersected. This could then be used to obtain the surface properties of that object.

In my implementation of each of the other techniques in this project, the reflected scenes are rendered using Unity's own lighting system to produce their final appearance. In my custom ray-tracing technique, access to lighting information was not available, so I implemented a Phong reflection model which approximated the appearance of the Unity system. The model can use the colour, texture and roughness of Unity objects as parameters. More complex lighting features could be added to the system in future, such as shadows, normal mapping, and alpha-blending, but these were not necessary for the technique to fulfil its purpose within this project. An example output from this technique is shown in Figure 3.11.

As the performance of this algorithm was too poor to be feasibly used interactively, I

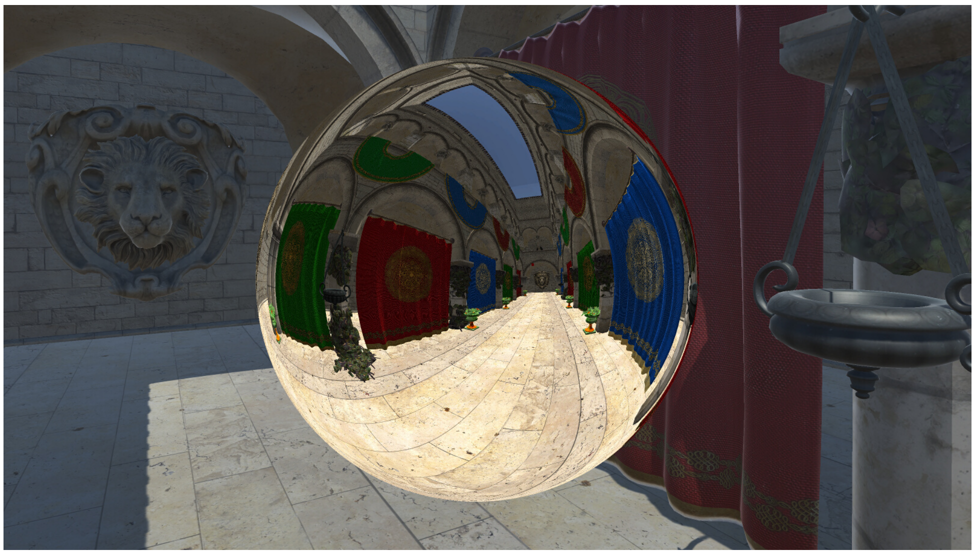


Figure 3.11: Accurately reflected geometry on a spherical surface, rendered using a CPU ray-tracing algorithm.

chose instead to allow ray-traced images to be directly written to a file using a control in the Inspector.

3.4 Summary

In this chapter I detailed the implementation process for this project. I implemented a technique for capturing cubemaps which I extended to capture depth cubemaps. I implemented a framework for managing different implementations of real-time reflections integrated with the Unity Editor and its user interface. I implemented the basic rendering technique for reflections using cubemaps, and extended this to several methods for parallax correction. I also implemented Planar reflections and a CPU ray-tracer to serve as a comparison to these techniques. Finally, I implemented the new reflection technique, involving ray-marching a depth cubemap. I discussed the challenges involved in optimising this technique and the possibility for future extensions.

Chapter 4

Evaluation

Following the implementation of the range of different reflection techniques, in this chapter I analyse how each performs in a range of constructed and “real-world” scenes. There are a range of different factors having effects on the performance of each technique, which are taken into account. In addition, each technique produces different types of visual artifacts or inaccuracies, which I classify and compare. I then go on to evaluate the success of the project itself, based on the Success Criteria in the Project Proposal.

4.1 Evaluation methods

4.1.1 Performance profiling

Frame-rate is commonly used as an indicator of the overall performance of a real-time system. However, in this instance this is too coarse-grained a metric, as many of the techniques are fast enough to have a minimal effect on frame-rate relative to other variables present in the runtime behaviour of a complex application such as Unity.

Instead, I measured the render time in milliseconds for each technique. For techniques running entirely on the GPU, this corresponds to the time for the draw call in which the reflective polygon/polygons are rendered. For algorithms also having a CPU component, in this case the Planar camera technique, I also took this into account, using Unity’s built-in profiling tools. The CPU ray-tracing technique which I used to produce accurate reference images was also tested with these tools, however, as discussed in Section 3.3.2, this technique is not intended to be competitive in performance terms for the purpose of this project.

GPU profiling was performed using RenderDoc, as mentioned in Section 2.2.1, to make repeated timings of the relevant draw calls. All performance profiling was performed on a stand-alone build of the project, to avoid the overhead of the Unity Editor influencing performance. The mean of 5 measurements was taken for each result, and a 95% confidence interval was computed.

The evaluation was performed on a PC with an Nvidia GTX 970 GPU, with the application running at a resolution of 1920×1080, on Windows 10.

4.1.2 Choice of scenes

The choice of 3D scenes on which the techniques are tested is crucial to a fair evaluation. Some techniques have a near fixed cost regardless of the scene. This includes the Uncorrected cubemap technique, and any method using proxy geometry for parallax correction.

For the Planar technique, as the scene is being re-rendered, its performance is directly correlated with scene complexity in terms of number of triangles (and the complexity of shaders present in the scene itself).

In techniques employing ray-marching, the novel DTPC technique and Screen Space Reflections, the relationship between scene complexity and performance is more subtle. The worst case in performance occurs when a ray takes the maximum number of steps before stopping. This occurs when the ray goes to infinity meaning that, somewhat counterintuitively, one worst case is when only the sky appears in the reflection.

The visual artifacts which occur with different techniques are also more or less visible depending on the specific layout of the environment, as is discussed later.

I chose a range of different scenes, including both freely available game environments and test scenes I generated myself, which aim to cover this range of factors. The pre-existing scenes chosen include the Sponza and Rungholt scenes [18], and a scene chosen from the Unity asset store[1]. More details of the scenes used can be found in Appendix A.

4.2 Performance evaluation

In this section I perform a quantitative evaluation based on the performance of each of the techniques investigated. I begin with some tests relevant specifically to the DTPC technique.

4.2.1 Ray-march steps

This test aimed to evaluate the effect of the number of ray-march steps on performance of the DTPC technique. The Sponza scene was used, and step length scaled inversely to number of steps to give the same maximum length for each ray. Figure 4.1 summarises the results, which confirmed the expected relationship between number of steps and draw time.

[1]`https://www.assetstore.unity3d.com/en/#!/content/52251`

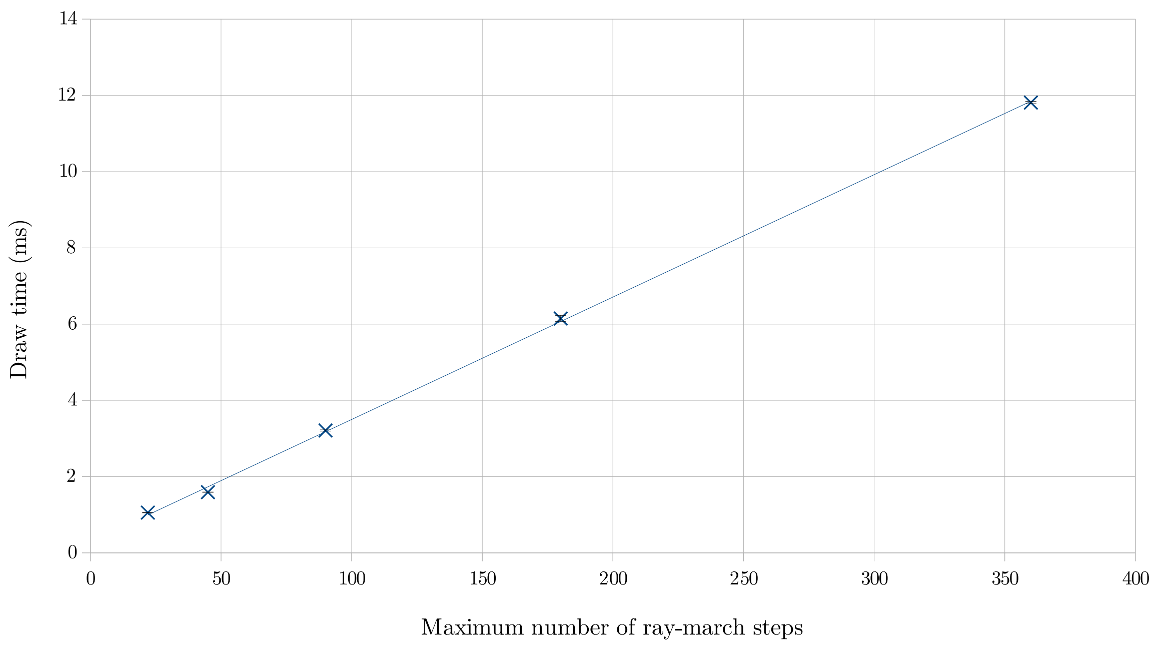


Figure 4.1: A graph showing the relationship between number of ray-march steps and performance.

4.2.2 Surface size

This test examined the relationship between number of reflective pixels and performance for the DTPC technique. As the algorithm runs per fragment a linear relationship was expected. Figure 4.2 shows that an approximately linear relationship was found. I would hypothesise that the deviation from this relationship is a result of different levels of cache coherence causing the cost of adding additional pixels to vary. This is in addition to the inherent errors in the GPU profiling method.

4.2.3 CPU overhead

The Planar technique has a performance overhead for setting up the reflection camera and associated render texture. This cost was found to be 0.34 ms on average. In comparison, for the cubemap-based techniques there is only a CPU performance penalty when the parameters of the technique are changed.

4.2.4 Scene comparison

This test aimed to give a general comparison of the performance of each technique across the range of scenes outlined in Section 4.1.2. An identical quad geometry was used as the reflective surface in each scene. For the DTPC technique, the number of step size used

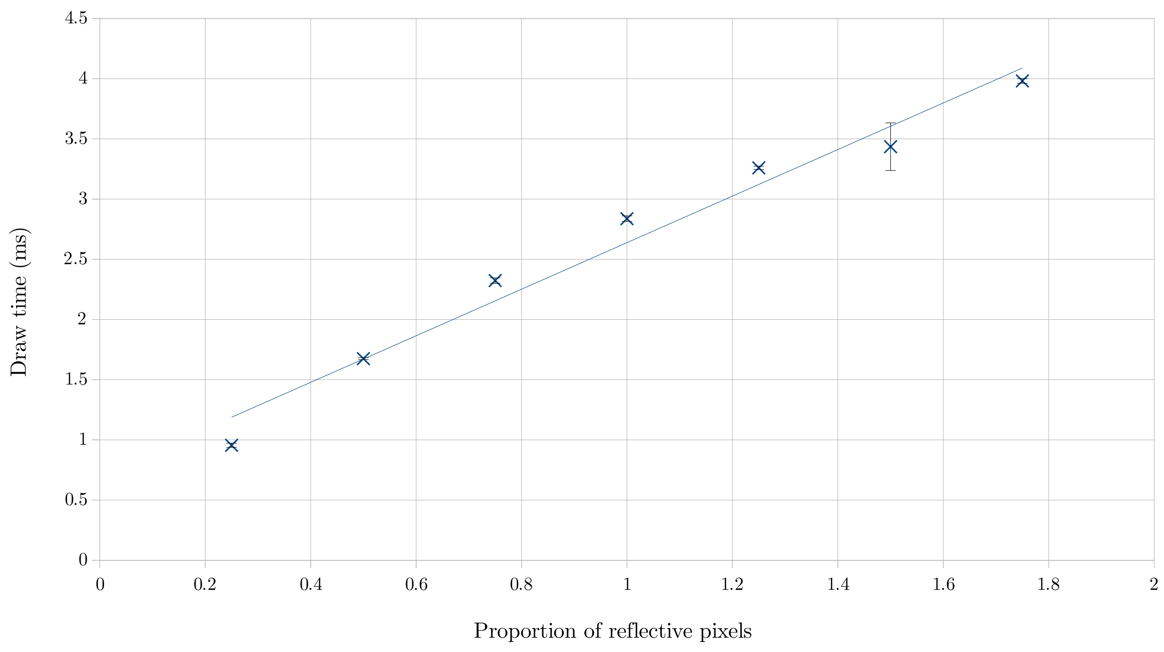


Figure 4.2: A graph showing the relationship between number of reflective pixels and performance. Number of pixels is given as a proportion of the pixels in a 934×934 pixels quad.

was chosen for each scene, aiming for the minimum number of steps that would give an acceptable result.

The performance of each method in each scene is summarised in Figure 4.3. The results confirm that in the proxy geometry case the cost is near constant, with different sizes and positions of the proxy geometry having little effect on performance. The performance of the Planar reflections are dependent on the number of triangles in the scene being reflected. This can be seen in the very high draw time for the Rungholt scene, which consists of a large and highly tesselated mesh. The performance of the DTPC technique is dependent on the number of ray-march steps performed for each pixel. This means that scenes in which the maximum number of steps is lower perform better, and also scenes for which much geometry is nearby allowing more ray-marches to terminate early.

The results show that as scenes become more complex and thus more expensive to re-render, the DTPC technique becomes more competitive in its performance. The exception to this was the Stanford bunny scene, in which the detailed geometry at a range of distances meant that a very large number of ray-march steps was required to obtain an acceptable result.

It is also worth considering that the DTPC technique is more widely applicable because it does not require the surface to be planar.

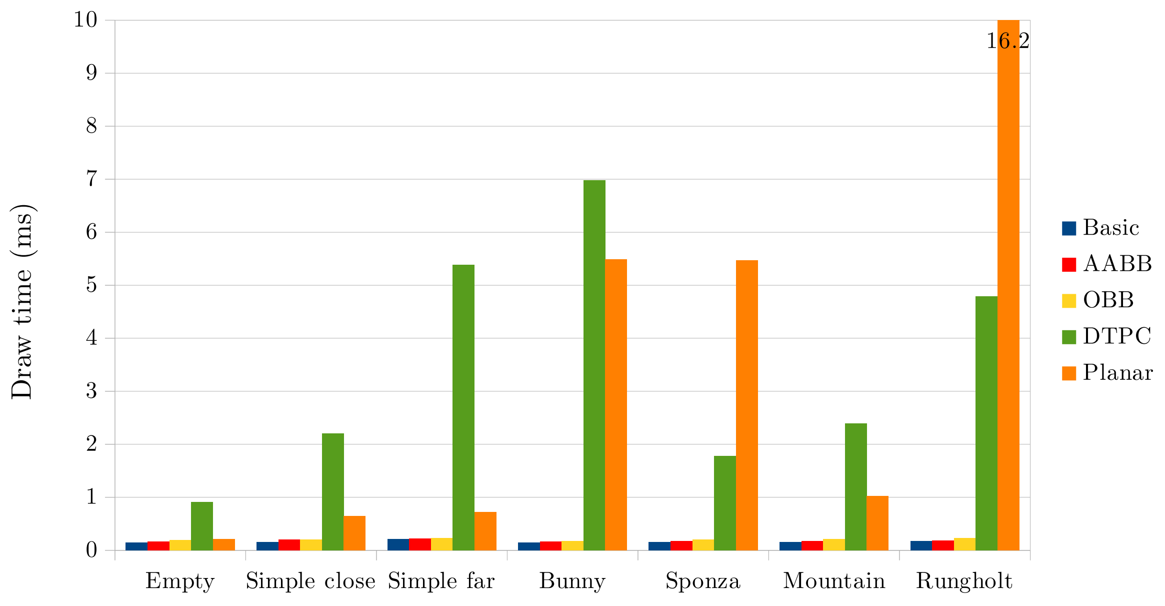


Figure 4.3: A chart comparing draw times for each reflection technique on each scene (lower is better).

4.3 Visual artifacts

Each technique implemented makes different assumptions about the scene, which introduce visual artifacts in the image produced. The techniques also have different limitations which may reduce their usefulness. In this section I categorise each of these and give examples. A summary of these artifacts and limitations is given in Figure 4.4.

- *Reflection unaffected by camera position (parallax error)*

  This is the error caused by the assumptions in the basic cubemap reflection technique, which is the motivation for the parallax correction techniques being investigated for this project. See Figure 3.6 for an example of parallax error.

- *Missing geometry occluded from sample point*

  In the case of cubemap-based techniques, the only information available about the scene is that which was captured in the cubemaps. Any parts of the scene which were occluded from this position therefore cannot appear in the reflected image. When using a proxy geometry, the approximation of the scene produced simply doesn't contain these areas. When using DTPC, holes are produced which can be filled using either information from the occluding pixels, or from a fallback cubemap texture, as in Figure 4.5.

- *Missing geometry occluded from camera*

  This is the major limitation of Screen Space Reflections, an alternative technique (see Section 2.1.4) which was not a part of this project.

	Uncorrected	Proxy geometry	DTPC	Planar	Ray-traced	SSR
Reflection unaffected by camera position (parallax error)	●					
Missing geometry occluded from sample point	●	●	●			
Missing geometry occluded from camera						●
Stepping artifacts			●			●
Objects distorted onto proxy geometry	●	●				
Reflected scene is static	●	●	●			
Surface must be planar				●		

Figure 4.4: A table which summarises the limitations and visual artifacts in each of the reflection techniques implemented. ● = the artifact or limitation is present.

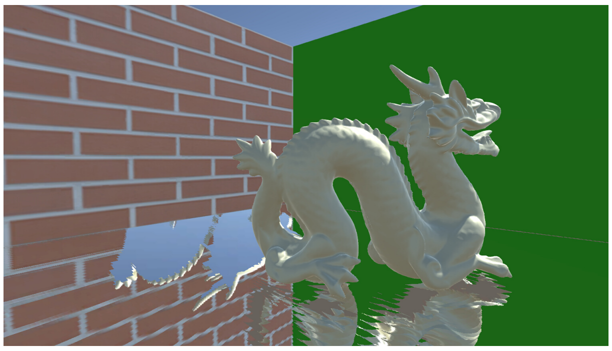


Figure 4.5: The dragon model placed close to the sample point causes a portion of the back wall to be occluded in the cubemap image. (When using the technique, objects close to the centre of the scene can be culled when capturing cubemaps such that a case like this would not usually occur.)

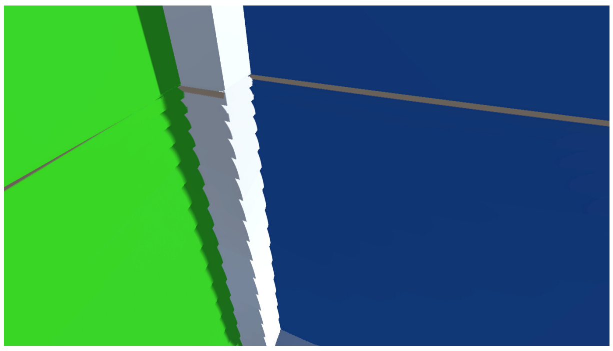


Figure 4.6: A reflection on a planar surface showing jagged edges due to the size of ray-march steps being used.

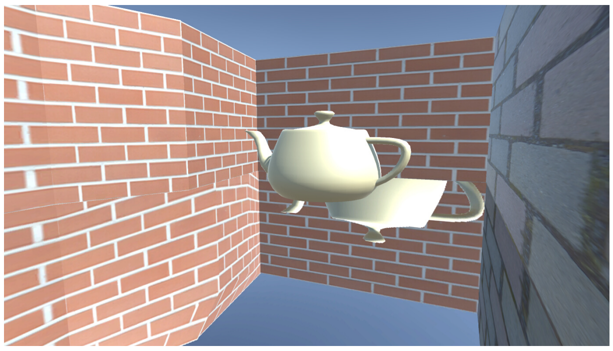


Figure 4.7: A planar reflective surface rendered with bounding box parallax correction. Parts of the scene which are not close to the proxy geometry, such as the slanted wall and teapot model, appear distorted in the reflection.

- *Stepping artifacts*

  This is a quantisation effect caused by the fixed steps taken in the ray-marching algorithm. It is most noticeable at the edges of flat surfaces, where a straight edge becomes jagged as a result, as shown in Figure 4.6. The significance of the effect is reduced by the introduction of an interpolation step, but cannot be removed entirely. In real-world scenes it is less visible as it is masked by the complexity of the geometry and textures in the scene, and by the properties of the reflective surface itself.

- *Objects distorted onto proxy geometry*

  Because the proxy geometry onto which the cubemap is being projected (such as AABB or sphere) does not exactly match the geometry in the cubemap image, there will always be some amount of distortion in the result produced by these techniques. Clearly this depends on how closely the proxy geometry and the scene match. See Figure 4.7.

- *Reflected scene is static*

  This is a limitation for any technique using a pre-computed environment map. Changes in the scene at runtime are not visible in the reflected scene. In contrast, techniques that use information about the current scene, including Planar and ray-tracing allow for dynamically changing scenes to be reflected correctly.

- *Surface must be planar*

  This restriction only applies to the Planar method because the camera setup only works given the assumption that the reflected image will be rendered onto a plane. This restriction is a significant reason for choosing other reflection techniques.

Each of these limitations is a trade-off in terms of increasing performance. As can be seen from Figure 4.4, the techniques that are most limited or inaccurate are those that generally give the best performance.

4.4 Summary

The results of both the quantitative and qualitative evaluations make it clear that there is no overall optimal technique for real-time reflection rendering. The most applicable technique depends on three factors: 1) the characteristics of the scene, 2) the characteristics of the reflective surface, and 3) the performance requirements of the application.

For the first point, the higher the complexity of the scene, the poorer the performance of the Planar technique, so the more appropriate a cubemap-based technique becomes. The DTPC technique can produce similar-looking results to the Planar technique which are competitive in performance for scenes with very large numbers of triangles. The layout of the scene also affects the usability of this technique, as scenes with detailed geometry across a wide range of distances will require a much larger number of ray-march steps to be rendered accurately.

For the second point, the accuracy of the reflection that need be produced depends greatly on the surface itself. It is easy to spot visual artifacts in a reflection rendered on a smooth or planar surface. Such errors are obscured on bumpy or curved surfaces. In these cases, only the general contents of the reflection (such as brightness and colour) need be accurate, and stepping and occlusion artifacts are less noticeable. This allows for techniques such as Proxy geometry parallax correction, or DTPC, to be used, and for fewer steps to be used in the latter. An example of this is shown in Figure 4.8.

Finally, developers may choose to sacrifice accuracy for greater performance on less powerful hardware, where even the Uncorrected cubemap approach may be the best choice.

In conclusion, every technique I implemented provides a useful solution to the problem for a particular context. An area for future work may be to investigate a means of classifying scenes to create a system which can select an optimal technique automatically given a scene and a set of requirements.

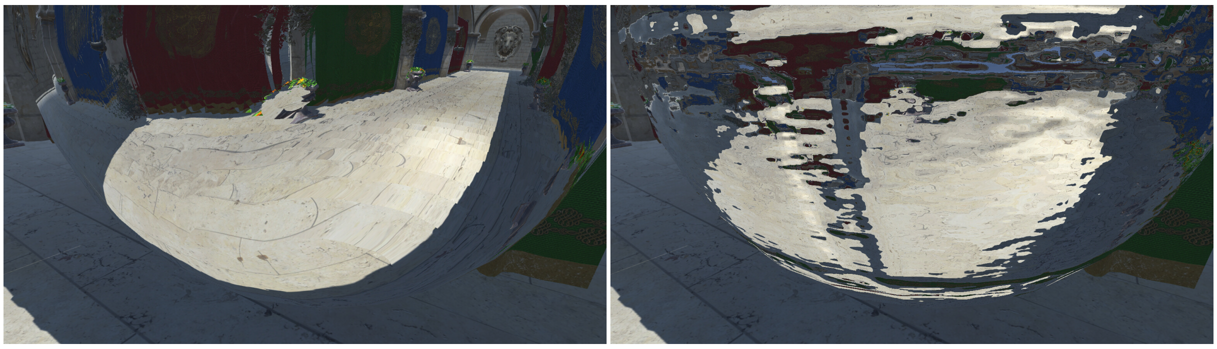


Figure 4.8: Visual artifacts due to too few ray-march steps (left) become more acceptable when the surface normals are less uniform (right).

4.5 Project evaluation

I now evaluate the success of the project itself, based on the fulfilment of the requirements specified in Section 2.3.

Requirements 1 and 2[2] were met in the successful implementation of all the cubemap-based techniques. These, and the other techniques implemented were designed to work within Unity and to be interchangeable by implementing a common interface, meeting Requirement 3. I was also able to successfully implement each of the alternative techniques I specified, meeting Requirement 4.

Following the evaluation which fulfilled Requirement 6, it is clear that every technique performs well enough to meet Requirement 5.

Of the extensions I gave, I achieved Requirement 7. The remaining two optional requirements were not met due to time constraints.

In addition to these specific goals, I feel that I achieved the overall goal of the project. I implemented a range of different techniques for rendering reflections in real-time, with a focus on investigating the potential of using parallax corrected cubemaps for this purpose. My evaluation has shown the merits of each of these techniques.

4.5.1 Challenges encountered

This project has given me an even greater awareness of the challenges inherent in programming for the GPU. Compared to CPU programming, the tools for debugging and profiling GPU code are far more limited. Finding the cause of bugs can be very challenging in a lot of cases, especially when working on an unknown algorithm when it is hard to know whether a bad result is a bug or just the output of an unsuccessful algorithm. This forced me to adapt to implementing algorithms in very small iterative steps, often outputting intermediate results as pixel values for analysis.

[2]In the electronic version of this document, the Requirement references are clickable.

In addition, in several instances a problem I encountered turned out not to be due to my code or the algorithm being implemented but due to some aspect of the Unity engine which I was unaware of. Documentation for most of the engine is good but some graphics-related topics were not well documented - and it is hard to find help when it is unclear what feature is causing a problem. Unfortunately the core of the Unity engine is closed-source so I was reliant on documentation and experimentation to understand its behaviour. These problems caused a few weeks of delays at more than one point in the project. Overall, I think that using Unity was the right choice for this project because it was quick to get started working on the core of my project, and the Editor tools available proved to be very useful in the Evaluation stage, however in future I would consider writing my own OpenGL application for graphics projects.

Chapter 5

Conclusion

In this project I set out to investigate approaches to generating efficient, accurate reflections in real-time rendering, with a focus on investigating the potential of cubemap techniques for this purpose.

I have implemented a range of techniques. These included uncorrected cubemaps, three variations of proxy geometry parallax corrected cubemaps, Planar reflections and a CPU ray-tracing implementation. In addition, I investigated and implemented a new technique using a cubemap depth texture to perform parallax correction using ray-marching, which produced competitive results.

I have shown that there is no overall optimal approach to solving the problem. The decision of which technique to use is a tradeoff based on a range of factors which I categorised in the Evaluation chapter. My implementations achieved acceptable performance for each technique, and I compared their visual results with that produced using the ray-tracing method.

In doing so, I completed all of the core requirements for the project, and one of the extensions, which I discussed further in Section 4.5.

5.1 Future work

The depth texture parallax correction technique which I implemented was capable of running in real-time and performed better than Planar reflections for some scene types. However, I believe that significant performance gains could be achieved using a data structure such as the minimum mipmaps proposed in Section 3.2.3.3. Implementation of an extended ray-marching shader operating on such a data structure is an interesting area for future work.

5.2 Final remarks

I believe the project was a success in introducing me to a wide range of topics in the field of real-time computer graphics, which built upon those covered in Part II of the Tripos. In re-implementing existing techniques I learnt a great deal about the challenges involved in programming for the GPU, in managing a reasonably large amount of source code, and in working with a complex existing system.

In investigating a novel reflection method, I believe I have implemented a technique worthy of further investigation that could prove to be useful in industry applications.

Bibliography

[1] Mark Claypool, Kajal Claypool, and Feissal Damaa. The effects of frame rate and resolution on users playing first person shooter games. In *Electronic Imaging 2006*, pages 607101–607101. International Society for Optics and Photonics, 2006.

[2] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[3] Stephen G. Lipson, Henry Lipson, and David Stefan Tannhauser. *Optical physics*. Cambridge Univ. Press, third edition, 1995.

[4] Naty Hoffman. Physics and maths of shading. In *SIGGRAPH: Practical Physically Based Shading in Film and Game Production*, 2012.

[5] Richard P. Feynman, Robert B. Leighton, and Matthew Sands. *The Feynman lectures on physics. Vol. 3: Quantum mechanics*. Addison-Wesley Publishing Co., Inc., Reading, Mass.-London, 1965.

[6] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[7] Han-Wei Shen. Real time reflections. `http://web.cse.ohio-state.edu/~whmin/courses/cse5542-2013-spring/18-reflection.pdf`, 2013.

[8] Bartlomiej Wronski. Assassin's Creed IV: Black Flag - road to next-gen graphics. *Game Developers Conference*, 2014.

[9] N. Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, Nov 1986.

[10] NVIDIA Corporation. Technical brief: Perfect reflections and specular lighting effects with cube environment mapping, 1999. `https://www.nvidia.com/object/feature_cube.html`.

[11] Gary McTaggart. Half-Life 2 / Valve Source Shading. *Game Developers Conference*, 2004.

[12] Iñigo Quilez. Terrain raymarching. `http://iquilezles.org/www/articles/terrainmarching/terrainmarching.htm`, 2016.

[13] Iñigo Quilez. Rendering Worlds with Two Triangles with raytracing on the GPU in 4096 bytes. `http://iquilezles.org/www/material/nvscene2008/rwwtt.pdf`, 2008.

[14] Sébastien Lagarde. Local image based lighting with parallax corrected cubemap. *Game Connection*, 2012.

[15] Art Tevs, Ivo Ihrke, and Hans-Peter Seidel. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *Symposium on Interactive 3D Graphics and Games (i3D'08)*, pages 183–190, 2008.

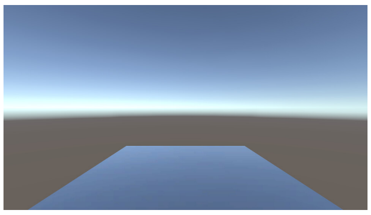
[16] Unity Technologies. Scripting API: Calculate Oblique Matrix, 2017. `https://docs.unity3d.com/ScriptReference/Camera.CalculateObliqueMatrix.html`.

[17] Eric Lengyel. Oblique view frustum depth projection and clipping.

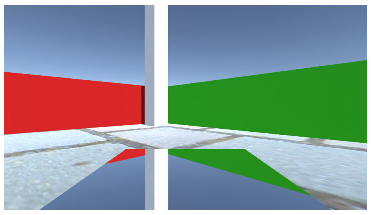
[18] Morgan McGuire. Computer graphics archive, August 2011. `http://graphics.cs.williams.edu/data`.

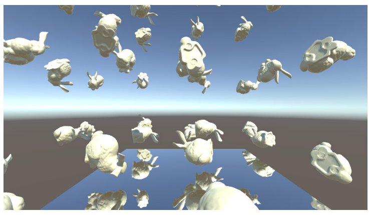
Appendix A

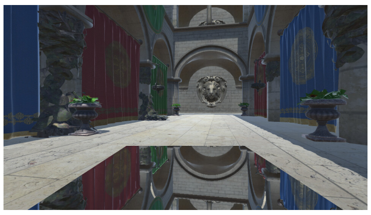
Test scenes

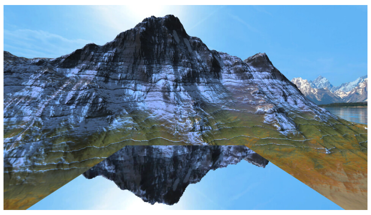

Empty

Simple close


Simple far


Stanford bunny


Sponza


Mountain

Rungholt

The images above show the scenes selected for the comparison in Section 4.2.4. The first four scenes were generated using the Unity Editor. The remaining three are based on existing 3D models. Details of the scenes, and ray-march parameters used, are given below.

- Empty: A skybox only. (Steps: 10)
- Simple close: Walls and pillars close to the camera. (Steps: 70)
- Simple far: Walls and pillars further from the camera, to give an idea of how the scale of the scene affects the techniques. (Steps: 100)
- Stanford bunny: 64 copies of the Stanford bunny model positioned randomly using a custom script, to give an idea of how techniques cope with complex and small

geometry. (Steps: 130)

- Sponza: A widely used and moderately complex scene representative of enclosed environments. (Steps: 45)
- Mountain: A simple but very large scene. (Steps: 40)
- Rungholt: A large outdoor environment which has a very large polygon count because of its voxel style. (Steps: 100)