MSc Dissertation Report

Granular Material Simulation

Submitted for the MSc in Computer Graphics Programming

September 2012

by

Braden Ellis
Abstract

This report presents an investigation of methods for simulating and rendering large systems of irregular or complex particles, such as grains, on the GPU.

Three different types of impostors are considered as well as standard billboard particles, all of which can be rendered by the accompanying program.

This report compares and evaluates these techniques and their behaviour on modern graphics hardware.
Table of Contents

1 Introduction ......................................................................................................................... 5
  1.1 Initial Brief....................................................................................................................... 5
  1.2 Context ............................................................................................................................. 5
  1.3 Aim and Objectives ......................................................................................................... 5
2 Project Background ............................................................................................................. 6
  2.1 Problem Context .............................................................................................................. 6
    2.1.1 GPU Simulation ......................................................................................................... 6
    2.1.2 Particle Systems ......................................................................................................... 6
    2.1.3 Billboards and Impostors .......................................................................................... 6
    2.1.4 Normal and Relief Mapping ...................................................................................... 7
    2.1.5 Related Work ............................................................................................................ 9
  2.2 Comparison of Technologies ........................................................................................... 10
    2.2.1 CUDA ....................................................................................................................... 10
    2.2.2 OpenGL ................................................................................................................... 10
  2.3 Comparison of Impostor Techniques .............................................................................. 10
    2.3.1 Dynamically-Generated impostors .......................................................................... 10
    2.3.2 Statically-Generated Impostors .............................................................................. 11
    2.3.3 Omnidirectional Relief Impostors ......................................................................... 11
    2.3.4 True Impostors ....................................................................................................... 13
    2.3.5 3-View Impostors ..................................................................................................... 13
3 Considerations .................................................................................................................... 14
  3.1 Legal Considerations ....................................................................................................... 14
  3.2 Ethical Considerations .................................................................................................... 14
  3.3 Social Considerations ..................................................................................................... 14
  3.4 Professional Considerations .......................................................................................... 14
4 Technical Development .................................................................................................... 15
  4.1 Expanded Specification ................................................................................................. 15
    4.1.1 Application Specification .......................................................................................... 15
    4.1.2 Particle System Specification .................................................................................. 15
  4.2 System Design ................................................................................................................ 16
  4.3 System Implementation .................................................................................................. 18
    4.3.1 Shader Objects ......................................................................................................... 18
    4.3.2 Storing Resources .................................................................................................... 18
    4.3.3 Storing and Rendering Objects ............................................................................... 19
    4.3.4 Particle Update ......................................................................................................... 19
    4.3.5 Billboard Rendering ................................................................................................ 21
    4.3.6 Dynamic Impostors .................................................................................................. 23
    4.3.7 Static Impostors ........................................................................................................ 24
Granular Material Simulation

1 Introduction

This report will discuss different methods of rendering grains and other high-detail objects as particles. It accompanies a program which will implement these methods in a 3D scene, and show that they can be used with techniques such as shadow mapping. The target audience is those involved in real-time graphics programming who wish to consider the ways of rendering particles in GPU-based particle systems.

1.1 Initial Brief

The properties of granular materials make their rendering an interesting challenge in real-time computer graphics. Grains are small enough to be represented as particles, yet often large enough to have visible details.

The application that accompanies this report will adapt many rendering techniques to GPU-based particle systems. They will be discussed and compared in terms of the algorithms they use, the resources they require from graphics hardware and their suitability for various situations.

1.2 Context

Real-time graphical applications strive for high rendering quality. In video games, for example, graphical realism is often a central selling point and vital to creating immersion. Detailed particle systems have many applications: games which take place in desert or Arctic environments would benefit from realistic-looking sandstorms or blizzards. Other real-time graphical programs could also benefit: an educational game about avalanche awareness may need to render a large number of rocks, while a real-time graphical simulation of the cardiovascular system might require particles with enough detail to distinguish different types of blood cell.

1.3 Aim and Objectives

The aim of the project is to investigate various real-time particle system rendering techniques by developing a program which implements them.

The initial specification was a program that included:

1. Particle systems simulated entirely on graphics hardware;
2. Collision detection and response between the particles and the terrain, as well as between the particles and objects;
3. The rendering of particles through billboards, dynamic impostors, and true impostors;
4. Shadows cast by objects and particles;
5. General graphics quality improvements: normal-mapping on particles and normal/relief-mapping on meshes, and a skybox.

An updated specification will be discussed in section 4.
2 Project Background

Real-time graphics applications strive to achieve both computational efficiency and visual detail. Much research has therefore been carried out to develop techniques that significantly reduce computation without sacrificing too much image quality. This section will describe the technology and methods on which the project is built, as well as related work.

2.1 Problem Context

2.1.1 GPU Simulation

The modern GPU is massively parallel, with a high number of shader cores (Nvidia 2006) capable of running thousands of lightweight threads in parallel (Buck 2007). In the field of general-purpose GPU computation (GPGPU), non-graphics algorithms have benefited from this parallelism, such as those used in protein folding (Luebke 2007), and collision detection (Lee et al. 2010).

Recent versions of graphics libraries allow shaders to output the results of their calculations back into GPU memory. OpenGL's transform feedback and DirectX's stream output are two such features.

Prior to this, general-purpose computation was still possible though the use of double-buffered textures. For example, a particle system might use one pair of textures to store the x, y and z position values and another to store the x, y and z velocity values (Kolb et al. 2004, Latta 2004). During an update pass, the fragment shader will read data from one buffer, update it through integration, and then write the new data into the other.

2.1.2 Particle Systems

Reeves introduced particle systems in 1983 as a way to represent fuzzier, non-mesh objects made of “clouds of primitive particles” which move and which have a lifespan, and are stochastic, in that that their positions at any given time cannot always be predicted (Reeves 1983).

A basic particle might have the following attributes:

- Position
- Velocity
- Acceleration
- Size
- Colour
- Transparency
- Age
- Shape
- Lifetime

An element of randomness should be present in order to improve the irregular appearance of the system. Random variation can be applied to the number of particles in a system, their initial velocity (Reeves 1983), and their size.

For a typical particle system, the physical simulation calculations are performed separately per particle, which makes them ideal for computation on a GPU.

2.1.3 Billboards and Impostors

In computer graphics, a billboard is a 2D object in a 3D environment that always faces the camera regardless of the position or orientation of either. A billboard with an unchanging
texture is suitable for objects whose appearance does not differ significantly when viewed in different angles. Billboards are also used to represent detailed objects, such as trees (Bao et al. 2009), at distances where the lack of perspective is less noticeable.

An impostor is a quad whose texture depicts a higher-detail object. They may be normal mapped, should the extra detail be required (see “Normal and Relief Mapping” further down in this section). Impostors are commonly used to portray objects that are far enough from the camera to be indistinguishable from the mesh itself. If this distance decreases, an impostor might be replaced by a higher-detail impostor, or the original object itself. Impostors are thus considered a level-of-detail (LOD) technique (Maciel et al. 1995).

The impostor’s texture itself can be created in real time in the application's graphics code (Schaufler 1995), or offline beforehand (Maciel et al 1995) and then loaded into graphics memory. Real-time impostors will hereafter be referred to as “dynamic” impostors, and those generated offline as “static”.

An impostor will need to change its texture when its current one no longer represents the correct view of the model. This process is often referred to as the impostor’s “regeneration”. An implementation of dynamic impostors might address this issue by testing how far the view vector (the vector between the quad’s centre and the camera position) deviates from the camera’s direction and the time since the impostor’s last regeneration. If either of these surpasses a threshold value, the impostor is regenerated (Davis 2006).

2.1.4 Normal and Relief Mapping
Normal mapping is a technique that uses normal data from a texture (called a “normal map”) when calculating the contribution of lights to the colour of a pixel. It is a type of bump mapping, which is the use of perturbed surface normals to give the object an irregular look (Blinn 1978). A normal map stores the \( x \), \( y \) and \( z \) components of a pixel’s normal in that pixel’s red, green and blue channels. Normal map data can be used in lighting calculations by transforming the vectors involved (light and view vectors) into tangent space, which is the coordinate space in which a face’s texture coordinates are specified.

Impostors have been extended using relief texture mapping (Oliveira et al 2000, Oliveira et al 2005). A relief texture can be seen as an extension of a normal texture, where surface data is stored as a depth map in the texture’s alpha channel. This data is used in the fragment shader, where a ray, based on the tangent-space vector between the viewer and the pixel, is cast “into” the texture data to determine what can be seen. Relief data can also be split into separate textures, such as when used to represent overlapping patterns (Policarpo et al 2006).
Consider the image shown in fig 2. A pixel’s intensity corresponds to its distance from the textured surface — lighter pixels (higher depth) represent areas lower underneath the surface, while darker pixels (lower depth) represent areas closer to the surface. The viewer at point $v$ is looking at pixel at point $p$, with $\vec{p}-\vec{v}$ being the tangent-space view vector. Given that the relief texture represents detail below the surface, the texture coordinates at point $p$ will not suffice.

Looking at fig 3, which represents the relief detail below the surface, it is apparent that it is actually the colour of texel $a$ that should be displayed at pixel $p$.

![Fig 2. Depth component of relief texture and tangent-space view vector](image1)

![Fig 3. Relief detail below surface](image2)

The pseudocode for the algorithm is in listing 1. The function takes as arguments the texture data, the texture coordinates of the pixel at $p$, and a delta vector that can be thought of as the ray’s slope. This latter value is used to decide how far along the $u$ and $v$ axes the ray moves with each step.

The function defines a number of linear and binary steps and a step size. It then performs a linear search, testing the texel depth at each step until it finds the texture coordinates of a texel whose intensity is lower than that of the ray; i.e., the approximate position of the point where the ray intersects the surface described by the relief texture.

A binary search follows to find a closer point to the intersection. It samples the depths of the texels around the initial intersection found by the linear search, continually reducing the step size until the loop finishes.
vec2 castRay(tex reliefMap, vec2 texCoord, vec2 delta) {
    int nLinearSteps = 50
    int nBinarySteps = 15

    float rayDepth = 0.0

    float stepSize = 1.0/nLinearSteps
    float texelDepth = texture(reliefMap, texCoord).a

    //linear search – note the loop does not break if the intersection is found
    //this is because it is a static loop whose number of iterations will be resolved
    //by the shader compiler
    for(int i = 0; i < nLinearSteps; i++) {
        if(texelDepth > rayDepth) {
            rayDepth += stepSize
            texelDepth = texture(reliefMap, texCoord + (delta * rayDepth)).a
        }
    }

    //"Rewind" to the point before the intersection
    rayDepth -= stepSize

    //binary search
    for(int i = 0; i < nBinarySteps; i++) {
        stepSize *= 0.5
        rayDepth += stepSize
        texelDepth = texture(reliefMap, texCoord + (delta * rayDepth)).a
        if(texelDepth <= rayDepth) {
            rayDepth -= stepSize * 2.0
        }
    }

    return texelDepth
}

Listing 1. Relief mapping ray cast function

2.1.5 Related Work
There are LOD techniques similar to impostors, such as “billboard clouds”, in which 3D models are represented by a textured set of planes (Décort et al 2003). In addition, there are alternatives that affect the detail of the geometry rather than replacing it entirely, such as adaptive rendering algorithms (Xia et al. 1997). Lower-level techniques, such as mip-mapping (Williams 1983), which work with a variety of higher-level methods, can also be considered.

Impostors have been applied to particle systems in an implementation by Charalampos, who also presents an alternative technique where a mesh object’s colour, normal and depth data are stored in cube maps made during the preprocessing stage (Charalampos 2008).

Similar work has been done in the area of cloud rendering, where particle systems are used with textures generated from functions (Harris et al 2002, Huang et al 2008).
2.2 Comparison of Technologies

There are a variety of technologies that can be considered in an investigation of GPU-based particle systems and impostors.

2.2.1 CUDA

In a GPU-based particle system, the movement simulation of each particle will be handled by graphics hardware. By themselves, physics equations alone have no graphical output, and can thus be considered to be general-purpose GPU computation. Nvidia’s CUDA is an architecture and programming model created to enable parallelizable algorithms to run on graphics hardware. CUDA has had success in optimizing N-body simulations (Nyland et al, 2007) and Navier-Stokes equations for fluids (Goodnight, 2007). It is demonstrably capable of working alongside graphics APIs such as OpenGL to simulate and render, at interactive rates, systems of particles which collide with one another (Green, 2010).

Despite its potential, CUDA will not be used in this project as the particle movement equations are sufficiently basic to take place in a GLSL vertex shader. Furthermore, use of CUDA would limit development and subsequent usage to computers with Nvidia graphics hardware.

2.2.2 OpenGL

OpenGL has been selected for both the simulation of the particles and the rendering of the scenes. A lot of documentation and code examples exist for it. It contains many of the necessary, more advanced features, such as transform feedback to update the contents of a vertex buffer object without leaving GPU memory, and geometry shaders, which are needed to expand point sprites into quads for many of the investigated impostor types. Most, if not all of the features implemented in the accompanying demonstration program could also have been implemented using Microsoft’s DirectX.

2.3 Comparison of Impostor Techniques

There are a wide variety of techniques that can be used to enhance the visual quality of impostors. Once again, they are discussed in this section in the context of GPU-based particle systems, and where the shader knows only about the vertex, particle or pixel it is currently operating on and cannot normally access any others.

2.3.1 Dynamically-Generated impostors

Dynamic impostors (Schaufler 1995) are created on-the-fly through render-to-texture operations. Previously, OpenGL would do this by rendering the image to a pixel buffer object, which would then be bound to a usable texture object (Wynn, 2002). These operations would often be window system-specific and require their own rendering context (and thus require use of costly operations such as wglMakeCurrent). A modern OpenGL implementation would use framebuffer objects (FBOs), which require a single rendering context regardless of their pixel format (Green 2005). An FBO can have a colour and depth buffer bound to it; both of which are needed to generate an impostor.

The benefit of a dynamic impostor is that a representation of an object can be displayed multiple times, while the object’s actual geometry need only be rendered once. The result of this, however, is that all impostors that represent the object will hold the same texture. This gives impostors the appearance of having undergone orthographic rather than frustum-based projection, with the only accurate impostors being those close to the centre of the screen. This is illustrated in fig 4. A workaround may involve rendering to several FBOs from different viewpoints, of which one would be chosen based upon its viewing angle, similarly to static impostors.
2.3.2 Statically-Generated Impostors

A “static” impostor is one that is not generated on-the-fly (Maciel et al. 1995). A static impostor's texture is chosen from a number of pre-generated ones. Generally, the texture chosen will be the one whose “direction” (the vector to the view position from which it was rendered) produces the smallest angle with the view vector.

The main advantage a static impostor has over a dynamic one is that, given a sufficient number of textures to choose from, it can always select a suitable one for its position. The downside of this is that a larger number of textures need to be generated for more complicated objects where “popping” between orientations will be more evident. This memory usage will increase when normal textures are used as well.

One solution to the popping effect is to replace the impostor with the model it represents at shorter distances. In a GPU-based particle system, however, this is significantly more difficult; a VBO cannot switch to another shader halfway through being rendered. The shader could choose the rendering method using a conditional branch based on the particle's distance, but most shader compilers will take away the branch and execute all of its stages, deciding on which result to use afterwards (Owens et al. 2007). While modern graphics cards support dynamic branching, in which a shader core may skip a branch's instructions if none of its threads enter that branch (Thibieroz 2006), this might involve having to sort the VBO's contents by distance on the CPU to ensure that the particles are spatially coherent.

2.3.3 Omnidirectional Relief Impostors

“Omnidirectional Relief Impostors” (ODR impostors) is an extension to impostors that uses relief mapping (Andújar et al, 2007). Like with statically-generated impostors, a preprocessing stage renders a set of textures taken from different viewpoints around the object. The set of textures will contain both colour and relief data. At run time, several impostors will be selected based on the viewpoint and positioned in such a way that resembles the structure of the object from which they were initially generated. Finally, in the fragment shader, a ray is cast into each relief texture to determine which part of the object is actually visible. Fig 5 illustrates what an ODR impostor relief texture looks like (normal data not shown).
While a relatively high number of impostor textures might be generated and stored on GPU memory (the paper suggests twenty), the number of impostors that will be used at any one time in the reconstruction depends on the complexity of the original object. The example videos (Andújar et al. 2007) show that a complex humanoid model can be reproduced with three impostors.

Although Andújar et al describe a large number of advanced optimizations and quality improvements, they also present and evaluate several basic options. This is an important factor in investigation, as it makes it easy to test if the basic technique is portable to the GPU, which must be done before any optimizations or improvements can be considered.

The paper outlines a number of improvements that can be made. For example, during the generation of the impostor textures, one can find the views close to each initial direction that contain the most “information content”. This technique, based on an information theory strategy (Vázquez et al. 2003) results in more coverage of areas with more detail, and less coverage of areas with less detail.

Frequently in an implementation of ODR impostors, several impostor quads might cover a single pixel. A decision can be made about which impostor will contribute; the paper suggests giving priority to the impostor whose direction is closest to the view direction. This is done by setting $\text{glDepthRange}(z_i, z_f)$, where

$$z_i = i \frac{f}{20}$$

where $i$ is the index of the impostor direction, sorted by their proximity to the viewing angle, from closest to farthest, and $f$ is an offset between 0 and 1. It also suggests using the depth value encoded in the relief map, but modified with a bias based on the angle between the impostor surface normal and the relief map’s normal (Andújar et al. 2007).
2.3.4 True Impostors

“True Impostors” (Risser 2008) is an alternative to ODR, and is also considered. It works by treating each billboard as a “window” to a texture of height fields which represent the object. This is done by treating the 2D texture as a 3D volume and treating the texture channels as four points along a new axis, $w$. A ray is then cast into the height texture, treating the latter as rotated based on the position of the billboard used as the window.

True impostors are likely to be faster than ODR impostors given that only one texture is considered, rather than the three or more needed to reproduce the image. In addition, a true impostor can store an object with concave or overlapping parts with as little as a single four-channel height field texture compared to the ODR impostor’s twenty.

True impostors cannot guarantee an artifact-free recreation of the object they represent, however; if one looks down the $w$ axis at a relief mapped quad, detail is visible that is generated from the full resolution of the $u$ and $v$ axes, while detail visible in a side-on view is limited by the limited resolution of the $w$ axis.

2.3.5 3-View Impostors

A technique similar to true impostors is the “3-View” impostor, which treats the object as a 3D volume based on three two-channel height maps; one for each coordinate axis. A voxel is considered to “contain” the object if the voxel, lies within the object in each view after being projected into the camera space for that view.
3 Considerations

3.1 Legal Considerations
This project’s legal considerations were those concerning the use of third-party libraries and art assets. Every resource used is either described as freely-available by the author, is in the public domain, or is available under a licence that allows non-commercial use. The authors of all resources used are credited in a text file in the project’s main folder.

3.2 Ethical Considerations
The main ethical issue is that of crediting the authors of techniques and algorithms. The project addresses this by citing, in this report, the papers in which the techniques and algorithms are introduced.

3.3 Social Considerations
This project is not a social study, and did not personally involve any third parties.

3.4 Professional Considerations
Effort was made to maintain a professional approach in all aspects of this project, including the production of high-quality deliverables, and in demonstrating the application.
4 Technical Development

4.1 Expanded Specification
The aim of the application part of this project is to simulate and render various types of particle systems in real time. Their visual quality will be consistent with a scene with several features of modern graphics applications such as height-map terrain, bump mapping, shadow mapping (Williams 1978) and a skybox.

ODR impostors were chosen over true impostors as the former builds upon statically-generated impostors, which were also being implemented: they share the pre-generated impostor textures, as well as the process of choosing suitable textures.

4.1.1 Application Specification
The expanded specification is as follows:

- The ability to store and render bump- and relief-mapped meshes. Two objects using the same mesh will render from the same VBO, reducing graphics memory usage. A model's vertices will be indexed, and thus will make use of the GPU's transform and lighting buffer for efficiency.
- The capability to store and render textures. As with model data, two objects using the same texture will refer to the same texture object in graphics memory.
- Lights, per-pixel shading and Blinn-Phong lighting (Blinn 1977) which will be performed in fragment shaders.
- A base particle system class for particles that are generated on the CPU and then sent to the GPU, where a vertex shader will update its position and velocity. Every particle in the system will remain in GPU memory for the running time of the application. The vertex shader will also handle the detection and response of the particles' collisions with the terrain and the mesh objects of the scene.
- The rendering of particles through normal-mapped point sprites, normal-mapped quads, dynamically-generated and statically-generated impostors as well as omnidirectional relief impostors.
- Shadows that can be cast and received by both mesh objects and particles.
- A skybox.
- The ability to read a “resources” config file that will specify the mesh and texture data to be loaded to GPU memory, as well as the number of FBOs to be created.
- The capability to read a “scene” config file that will determine the objects to be displayed, the mesh and textures to be used in their rendering, as well as properties such as their position. The properties of the particle systems will be specified here.

4.1.2 Particle System Specification
The particle system types to be implemented by the program are as follows:

- Point sprite particles
- Quad particles, which start off as points and are expanded in the geometry shader
- Dynamic impostor particles
- Static impostor particles
- Omnidirectional relief impostor particles

The application will implement these particle systems through individual particles, which are stored in GPU memory; and particle “emitters”, collections of attributes that remain in the CPU portion of the code.

Particle emitters specify the following characteristics of the particle system:
Granular Material Simulation

- Position. This refers to the position of the unmoving particle emitter as well as the initial position of each particle.
- Velocity. This can be thought of as the emitter’s direction. Upon generation, a particle will apply a small random variation to this value and use the result as its initial velocity. The emitter itself does not move.
- Acceleration. Like velocity, this is a particle-specific value which can be given random variation.
- Number of particles.
- Colour texture. This system-wide value refers to the texture of all particles in the system. For static and ODR impostor systems, this attribute is a reference to a collection of pre-generated impostor colour textures.
- Normal texture. This is the texture that a non-impostor particle system will use to apply bump mapping to its particles. Like the colour texture attribute, static and ODR impostors will instead refer to a collection of pre-generated impostor relief textures which contain both normal and depth data. The depth data is not used by static impostors.
- Directions. This attribute is only used by static and ODR impostors, and refers to a collection of vectors which specify the normals for each impostor texture.
- FBO. This attribute is unique to dynamic impostors. It is an object of a separate class that stores references to the data needed to create impostors on the fly. These include the model to render, its colour and normal textures, and the locations on GPU memory of the colour and normal textures it has created.
- Size. Like velocity and acceleration, each particle can be sized according to a random variation of this value.
- Lifespan. This is the lifespan of each particle within the system rather than the emitter itself. Once a particle has reached the end of its lifespan, it will be reborn.
- Frequency. This is the time interval in seconds between particle emissions.
- References to the “ping-pong” VBOs which store the particle data.

The particles themselves will have the following attributes:
- Current position. Before particle birth and after death, this is the same as the emitter’s position.
- Current velocity.
- Initial velocity. After a particle dies it is re-emitted, and its velocity is reset to this value. As each particle’s velocity value is unique due to the random variation, it cannot be set to the emitter’s velocity vector. An alternative would be to take the emitter’s velocity and apply a new random variation within the shader.
- Start time. Prior to the initial emission of the ith particle in a system, this value will be $s + if$, where $s$ is the application’s start time and $f$ is the system’s emission frequency. Once a particle dies, its time of death will be stored at the start time, and it will be reemitted in the next frame.
- Age. This value grows with the frame’s timestep once the particle has been launched, and is set to zero when the particle dies.

4.2 System Design

The initial specification and design can be found in the appendix. Notable changes were made to both the design and specification throughout the later design stages as well as development. The clearest of these are visible in the simplified class diagram in fig 6.
The earliest deviations from the initial design involved the external libraries. SFML was chosen over FreeGLUT as it provides many necessary additional features such as image loading and timer classes. SFML is free for commercial or non-commercial use, and all of the external libraries it uses are used under the BSD, LGPL, zlib/png and GPL licenses, or are public domain.

GLM, a vector mathematics library designed to resemble GLSL’s maths functions, is considered in the original specification in the appendix, and is present in the updated design. It is used in the majority of the application’s classes, and is omitted from the updated UML diagram for clarity. Many vector calculations will be made on both the CPU and GPU parts of the program, and use of GLM will result in similar syntax throughout the source. GLM is available under the MIT license.

Boost is also considered in the initial design, but is not used, as the functionality that was desired is provided elsewhere; timing classes are present in SFML, and hash maps are provided by C++’s unordered_map class.

GLEW is used in the project, and is available under the MIT license. Blender and GIMP are used to make models and textures.

The second significant change is the hierarchy of the objects. In the early stages of implementation, it was found that lights, particles and mesh-based objects had more differences than similarities, and as a result, Object3D was changed from a base class to the class for mesh-based objects. The updated design uses inheritance to derive ImpostorParticleEmitter from ParticleEmitter, on which it builds, adding functionality for static, dynamic and ODR impostors.
The third major structural change in the updated design is that there is no longer a resource of scene manager, and that singletons are not used. Resource-loading functions were deemed more suitable for the renderer, as all of them were concerned with loading graphics asset data onto GPU memory. The scene manager was dropped as it was deemed that holding more than one scene at runtime was an unnecessary feature. Finally, the `RenderManager` singleton was replaced with `Renderer` as it is not essential for the class to be global.

4.3 System Implementation

4.3.1 Shader Objects

The focus of this project is on graphical techniques, and as such, most of the features involved require one or more shader programs. So that shader code is reusable, shaders are implemented as objects. A shader object contains:

- A handle to an OpenGL program object, vertex shader, geometry shader and fragment shader;
- Flags that specify whether the shader has been successfully linked and validated;
- A string into which any compilation or linking errors will be copied;
- An unordered (hash) map, where strings holding the names of GLSL attributes and uniforms map to their locations; and
- Functions to load, compile and link shaders, and to retrieve their logs if any of these steps were unsuccessful.

A shader configuration file (separate to the “resource” and “scene” config files) contains, for each shader, a string label that acts as the shader’s “name”, the locations of the vertex, geometry and fragment source files (geometry and fragment shaders can be ignored for shader programs which do not have any) and the names of the attributes and uniforms the shader uses.

The main benefit of this is that the `Renderer` class can load and compile a shader with only one function call, `loadShader`, which takes as arguments the location of the shader configuration file, the label of the shader within the file, and a reference to the shader object.

4.3.2 Storing Resources

The `Renderer` class has unordered maps for meshes and textures. For both, the key is a string which describes the resource and the mapped value is the resource’s OpenGL location or a container class; either an object of the `Mesh` class (which holds references to a vertex and index buffer) for mesh data or a texture object for texture data. The `Object3D` and `ParticleEmitter/ImpostorParticleEmitter` classes hold strings which map to these values and determine the resources they use. For example, a model of a mongoose might be stored in a VBO inside a `Mesh` object associated with a key called “MongooseMesh” and its colour texture might be stored in a texture object with a key called “MongooseTex”. Any `Object3D` that represents a mongoose will have “MongooseMesh” as its `m_meshKey` variable and “MongooseNormal” as its `m_texKey` variable.

Like colour textures, normal and relief maps are stored in the texture vector.

The terrain mesh is generated from a height map when the application starts. This is done in the `Mesh` class function `createFromHeightMap`. This function takes in an argument of type `HeightMap`, a class that contains the height map’s attributes that were read in from the resource list config file, such as height and spacing, as well as a vector of pixels. It uses this data to create a mesh with a number of vertices equal to the height map texture’s pixel count. Vertices are spaced apart equally by the height map’s `m_spacing` value, and their height is equal to the height map’s `m_heightScale` parameter multiplied by the source pixel’s alpha value.
4.3.3 Storing and Rendering Objects

The `scene` class contains separate vectors to store mesh objects, lights, and each type of particle emitter. The `Renderer` class, whose `drawScene` function takes a reference to the program’s single scene object, draws each type of object by first activating the shader for that object type, and then iterating through the vector, drawing each element. The grouping of object types this way reduces the need to switch shaders.

Each object type has its own drawing shader, which writes to the screen frame buffer, and its own shadow shader which writes to one of the shadow depth buffers stored in `Renderer`’s `m_meshShadowFBO` and `m_particleShadowFBO` variables. An object’s shadow shader is similar to its standard render shader, but lacks access to the light objects, as well as the texture fetches for normal map textures (unneeded as shadows do not reflect light) and the shadow depth map (as it will be writing to it). All of the particle system shadows require access to the colour texture maps as it is the texture used by a billboard that defines its silhouette.

A diagram illustrating the order of operations of the `drawScene` function follows fig 7.

![Diagram](image.png)

Fig 7. The `drawScene` function

4.3.4 Particle Update

The `Renderer` class stores a vector of unsigned integer pairs. These are the locations of the two VBOs used to perform transform feedback. After having generated the particles, the application requests space for two VBOs rather than one, and assigns the locations to the emitter in question. It also assigns them to the aforementioned vector. This is so the
*Renderer* class, which handles the majority of the OpenGL functionality, can destroy the VBOs when the application exits.

The *ParticleEmitter* and *ImpostorParticleEmitter* classes hold an integer variable called `m_currentTF` which identifies which of the two buffers is to be written into during the update stage. After the particle system is rendered, this variable will point to the other buffer. This technique is called “ping-ponging” and is illustrated below fig 8.

![Ping-ponging Diagram](image)

Fig 8: ping-ponging

The benefit of this technique is that it enables us to update the particle’s age so that we can destroy and re-emit it once it has passed its lifespan. This means that we can reuse particles which are likely to be no longer moving or which have moved too far away to be visible. In addition, storing a particle’s state at the most recent timestep makes it much easier to calculate collision detection and response.

The shader object, `m_pupShader`, comprises a single vertex shader, `particleupdate.vert`, with no geometry or fragment shader. As the name suggests, the shader’s function is to update a particle’s position, velocity and age. It does the first two by performing Runge-Kutta integration, and the last by incrementing the current age of the particle with the timestep.

The shader also performs collision detection and response, in the following ways:

- **Bounding box.** Up to four mesh objects can be defined in the “scene” config file to have bounding boxes of varying sizes.
- **Bounding sphere.** Four objects can be defined to have bounding spheres of varying radii.
- **Height map.** The shader can access the texture object that contains the height map data of the terrain.

A particle is considered to be inside a bounding box if it is behind each of the planes containing the box’s faces. Should this be the case, the shader determines which of the planes the particle is closest to, and reflects the particle about the face’s normal.

Bounding sphere collision is detected using the shader’s `sphereIntersect` function using the test

\[ c = (||\vec{v}||)^2 \leq (r_s + r_p)^2 \]
where \( \vec{v} \) is the vector between the centres of the sphere and particle, and \( r_s \) and \( r_p \) are the radii of the sphere and particle. Should a collision be detected, the particle will be reflected about \( \vec{v} \) (after it has been normalized).

To detect particle-terrain collisions, the shader must first determine the height of the ground at the particle’s position. It does this by taking the particle’s \( x \) and \( z \) coordinates and scaling them by \( s \) in order to get the texture coordinates of the particle’s position.

\[
s = \frac{1}{l \times r}
\]

\( l \) is the length of the texture in pixels and \( r \) is the spacing ratio used in the initial creation of the terrain mesh.

Once the correct texture coordinates have been determined, the shader fetches the texture value at those coordinates, and scales it up by the height factor initially used to scale the terrain mesh. This returns the correct height of the terrain. If the height of the particle is lower than that of the terrain, it is reflected.

If any sort of collision is detected, the particle’s position is reset to the position it held before integration was applied.

4.3.5 Billboard Rendering

Standard, non-impostor billboard rendering is performed by two shaders, \( m\_prendShader \) and \( m\_prendShaderSmall \). The former of these expands the particles into quads, while the latter renders them as point sprites.

Splitting the functionality between two shaders was necessary because many OpenGL driver implementations set a limit to the size in pixels of a point sprite.

The \( m\_prendShader \) object has a vertex, geometry and fragment shader (\( particlerender.vert, .geom \) and \( .frag \)). The vertex shader only transforms the point to view space. The geometry shader then expands it into a quad made from a four-vertex triangle strip. The new vertices are positioned at half-lengths around the original point, thus forming a square with length and height equal to the particle’s size attribute, and with the original point position in the centre.

As the new quad is a billboard, these new vertices do not need to be rotated. The positions they are generated with are left as their camera-space coordinates. Appropriate texture coordinates (ranging from 0,0 to 1,1 for the bottom left and top-right vertices) are also generated for each new vertex. The result of this process can be seen in fig 9.

The geometry shader also calculates the shadow map coordinates for each vertex. It thus, for each new vertex, takes the untransformed position of the original point, transforms it into the light’s view space, and then once again translates it by the particle’s half-length to its new position.
Normals are not generated at this stage, as the quad will have a normal map applied, and as it is always facing the camera, its view vector is already in tangent space.

The geometry shader also calculates the shadow map coordinates for the quad. For each new vertex, it takes the untransformed position of the original point, transforms it into the light’s view space, and then once again translates it by the particle’s half-length to its new position.

The fragment shader applies the texture, normal map, and calculates lighting effects.

The point sprite shader, *particlerendersmall*, is significantly more basic. It sizes the sprite according to the viewport width and the distance. The fragment shader textures the sprite using the built-in `gl_PointCoord` variable.

---

Fig 9. Newly-created billboard

Fig 10. Billboard particles
4.3.6 Dynamic Impostors

Dynamic impostors need to be declared in the resource configuration file, giving a string to act as the name of the FBO, and specifying the mesh, texture and normal texture that will be used in the rendering of the impostor. An FBO class is made which then stores these. The FBO class also holds references to the three textures to be generated: m_texID for the colour texture, m_normID for the normal map and m_shadowDirTexID for the impostor texture generated from the point of view of the sun. The generateImposterParticleTexture function of the Renderer class generates these three textures.

The size of the impostor texture in pixels is specified in the scene file, but the size of the image depends on the mesh used. When a mesh is loaded, its extents (the distances between the farthest vertices along the x, y, and z axes, with “maxExtent” being the length of the longest diagonal in the cuboid with the three extents as its dimensions) are calculated and stored in the Mesh object that is created. During the dynamic impostor-generation stage, an orthographic projection is used which uses these extents as its dimensions, and the camera’s position vector is normalized and multiplied by maxExtent. As the object to be rendered is positioned at (0, 0, 0), this will result in the camera being positioned maxExtent away from the object to be rendered. This approach results in the loss of some perspective detail, but is an easy way to ensure the entirety of the object is rendered to the impostor texture.

The shader object used to draw dynamic impostors is the same as the one used to draw billboard quads, m_prendShader. The colour and normal texture used are those generated by the FBO.

![Fig 11. Dynamic impostor particles](image)
4.3.7 Static Impostors

The application does not render static impostors as billboards, but rather as quads oriented in the direction of the view from which the impostor texture was initially rendered. This effect is unnoticeable except for large particles at a small distance, and there for code reuse purposes: they share code with the implementation of ODR impostors, where this characteristic is important.

Static impostor textures need to be rendered beforehand from a selection of angles. In this implementation, 80 colour textures and 80 relief textures are rendered from 80 evenly-spaced directions. This is performed by the application GenImpostors which was developed internally for this project. GenImpostors produces relief maps rather than normal maps, but static impostors ignore the alpha channel used to store depth. It also produces a list of 80 directions and perpendicular "up" vectors, which are loaded into the ImpostorParticleEmitter object on program startup.

Similarly to the generation of dynamic impostors, GenImpostors positions the mesh in the centre of the scene, and renders it to a texture with orthographic projection. The camera’s 80 positions are the normalized normals of an 80-faced icosphere (made in Blender), multiplied by maxExtent.

Static impostors are rendered by the m_prendStaticImpostorShader object, which contains a vertex, geometry and fragment shader (impostor_particle_render_static.vert, .geom and .frag). The vertex shader iterates through an array of the 80 direction vectors (passed in as a uniform) and compares each against the vector between the camera and the particle, returning the index of the one which produces the smallest angle.

The geometry shader is similar to the geometry shader used to make the billboards and dynamic impostor particles. A new vector, "side" is generated by taking the cross product of the "up" and "direction" vectors whose indices were selected in the vertex shader. The new vertices are translated along the "up" and "side" vectors to give the impostor its orientation. In addition, as they are not billboards, they must be individually translated by the worldview matrix into camera space.

Tangent space calculations must also be done, as the quad is no longer guaranteed to be facing the screen. The normal used to make the tangent matrix is the direction vector, the tangent is the side vector and the bitangent is the up vector. All are transformed by the normal matrix (the transpose of the inverse of the worldview matrix) before being used to generate the tangent matrix. A tangent space view vector is generated for each of the new vertices.

The fragment shader applies the texture, normal map and shadow.
4.3.8 Omnidirectional Relief Impostors

The implementation of ODR impostors is similar to that of static impostors. They also make use of the 160 colour and relief textures made by the GenImpostors program. The m_ODRimpostorRenderShader object handles the rendering of ODR impostors.

The vertex shader, ODRimpostorrender.vert, iterates through the array of directions. This search is made three times to find the indices of the first, second, and third most suitable impostors. It is these that will be used to regenerate the detail of the object. The shader passes on these indices to the geometry shader, as well as the centre positions of the new quads to be generated. A new position is equal to:

\[
\vec{p'} = \vec{p} - (\vec{d} \cdot s \cdot 0.5)
\]

where \(\vec{p}\) is the position of the particle, \(\vec{d}\) is the direction vector of the impostor, which is normalised by GenImpostors on its generation, and \(s\) is the particle's size.

The geometry shader, ODRimpostorrender.geom, creates three quads and three different tangent-space view vectors. This is done by giving the shader an input layout qualifier of \textit{layout (points, invocations = 3)} in, with "\textit{invocations = 3}" telling the geometry shader to output 3 primitives. The shader thus decides which of the quads to draw based on the value of \textit{gl_InvocationID}.

The fragment shader, ODRimpostorrender.frag, is where the ODR impostor particle most significantly diverges from the other types of particles. The pixels of the ODR impostor are not merely sampled from a texture - relief mapping is used to regenerate the detail of the actual object that would be visible from the current view. See the "Relief Mapping" section in chapter 2 for more detail. Fig 13 shows a comparison between the output of the shader with relief mapping off, and the output with relief mapping on.
Fig 13: ODR impostors with relief mapping off (left) and on

The fragment shader has the index of one of the impostor textures chosen earlier by the vertex shader, as well as the tangent-space view vector for that quad. It then uses the `castRay` function with the view vector to determine the location of the pixels which would actually be visible.

An overview of the algorithm can be seen in the “Normal and Relief Mapping” part in section 2. The ODR impostor variant, whose source can be found in the appendix, is slightly different; as well as returning the location of the first intersection, it returns the depth of the closest point along the ray before the intersection. This is done in the binary search loop: if it moves forward slightly and detects an intersection, it stores the depth of the current point as the “best depth in”; if it does not detect an intersection, it stores the depth of the current point as the “best depth out”.

If the difference between these two depths is too high, the fragment will be discarded. This is to reduce “wall” effects – if the “in” depth is 0.5 and the “out” depth is 1.0 and the impostor is viewed from an angle, a “wall” between these two depths will be visible. In this project’s implementation, the threshold difference is 0.04. Fig 14 shows a comparison between an impostor which culls these walls, and an impostor that doesn’t.

Fig 14. ODR impostor with wall culling on (left) and off
4.4 Relief Mapping
Before ODR impostors were implemented, the basic relief mapping algorithm had to be tested. Details about the algorithm can be seen in section 2.

![Relief-mapped cube](image)

Fig 15: Relief-mapped cube

4.5 System Testing

4.5.1 Test Specification
The efficiency of various types of particle will be tested by rendering them in varying numbers in a specially-created scene.

The test scene includes:
- A 1024 x 1024 shadow map to which the particle shadows are rendered
- A height map mesh with 65536 vertices
- A 1024 x 1024 texture for the terrain
- A skybox made from six 1024 x 1024 textures

The tests are made by positioning the camera in the center of the map and facing the “mouth” of the emitter, such that the particles are fired towards the camera. The emitter is spaced at (0.0, 4.0, -6.0) and has a velocity is (0.0, 0.0, 10.0). The particle lifespan is set to 0.35 seconds so that the particle will die and be reborn before it can move out of the camera’s view.

For each particle type, separate tests are made for emitters with 20, 80, 320, 1280, 5120, 20480 and 81920 particles.

The point sprites tested have the following attributes:
- 512 x 512 texture
- 512 x 512 normal map
- Size of 0.05
The quad billboards tested have the following attributes:

- 512 x 512 texture
- 512 x 512 normal map
- Size of 0.05

The dynamic impostors tested have the following attributes:

- Generated from a mesh whose VBO has 1517 vertices and 7572 indices
- The mesh has a 512x512 texture and a 512x512 normal map
- The texture and normal map rendered to are 512x512 in size
- Impostor has a size of 0.5 (appears roughly as large as the 0.05 billboards from the previous tests)
- Regenerated each frame

The static impostors tested have the following attributes:

- 512x512 colour and normal textures
- Size of 0.5 (appears roughly as particles in the previous tests)

The ODR impostors tested have the following attributes:

- 512x512 colour and normal textures
- Size of 0.5 (appears roughly as particles in the previous tests)
- Raycasting function with 50 linear steps and 15 binary steps

4.5.2 Test Results

The results of these tests are shown in table 1 and fig 16. Data is missing for 5120, 20480 and 8190 ODR impostor particles, as the application became too slow to be reliably tested.

<table>
<thead>
<tr>
<th>Number of Particles</th>
<th>Point Sprites</th>
<th>Quads</th>
<th>Dynamic Impostors</th>
<th>Static Impostors</th>
<th>ODR Impostors</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>122.3137388</td>
<td>122.84322</td>
<td>115.071711</td>
<td>115.1233927</td>
<td>38.06873711</td>
</tr>
<tr>
<td>80</td>
<td>122.7411445</td>
<td>119.99886</td>
<td>113.1583683</td>
<td>99.883482</td>
<td>18.82981665</td>
</tr>
<tr>
<td>320</td>
<td>120.4598206</td>
<td>127.11006</td>
<td>116.4918504</td>
<td>95.47669333</td>
<td>2.830175307</td>
</tr>
<tr>
<td>1280</td>
<td>115.3983846</td>
<td>105.96655</td>
<td>91.9470634</td>
<td>60.85373919</td>
<td>1.744628381</td>
</tr>
<tr>
<td>5120</td>
<td>107.6212018</td>
<td>100.22546</td>
<td>89.00316654</td>
<td>23.41612903</td>
<td>6.447608868</td>
</tr>
<tr>
<td>20480</td>
<td>96.24855374</td>
<td>98.335232</td>
<td>56.85426644</td>
<td>6.447608868</td>
<td>1.512946253</td>
</tr>
<tr>
<td>81920</td>
<td>77.85663722</td>
<td>56.962977</td>
<td>21.42970413</td>
<td>1.512946253</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Results of testing
Fig 16. Results of testing
5 Evaluation

5.1 Efficiency
The results in section 4 show that while point sprites and quad billboards cope well with larger systems, the impostors do not.

Dynamic impostors are expected to be slower due to the render-to-texture operation that is performed each frame. The performance gap becomes noticeably larger at 2048. One reason might be the larger number of pixels; the quads had to be made much larger for the particle to appear the same size, leading to a greater workload for the fragment shader.

The static impostors are relatively slow. One possible reason is that the vertex shader must iterate through 80 directions to find the one closest to the camera direction.

ODR impostors are by far the slowest. The vertex shader must loop through the array of 80 directions three times, and the geometry shader must create three new quads (twelve new vertices). The fragment shader must perform ray casting on the pixels of each of these impostors.

An alternative to looping through the direction array would be to store the data inside a texture. A texture fetch can take thousands of cycles (Fatahalian 2010), but may be faster than the comparisons in the current implementation. The direction indices could be stored in a 2D texture; a direction's index can be found by treating its x and z coordinates as u and v texture coordinates, and accessing the texture value. For sets of views where two directions share the same x and z values, the two y values can be stored in the red and green channels, and their indices in the blue and alpha channels. This would reduce 80 iterations to a texture fetch and a comparison to determine if the view vector's y coordinate is closer to the red or green value.

5.2 Visual Quality
The implementation of ODR suffers from many visual artifacts, which can be seen in fig 17. For most angles, the whole view of the object will be reconstructed, but fails for views that show a lot of detail in the “inner” areas of the mesh; such as the underside of the rolled-up armadillo.

The most likely reason is that the views used provide insufficient coverage of the more covered areas, as is visible in fig 17, where the area that is not reproduced is an area that would be covered by the armadillo's tail in other views. A possible solution is discussed under “Further Development” in section 6.
The other major artifacts are caused by the “wall” effect discussed under “Omnidirectional Relief Impostors” in section 4. Though the majority of the wall is culled, several bands remain. The culling threshold can be decreased so that smaller “jumps” in depth are discarded, but this will also lead to the loss of pixels in areas where significant depth differences are part of the object detail. This can be seen in fig 18, where the threshold has been lowered from 0.04 to 0.02. Note the artifacts on the side of the armadillo’s tail and underneath the chin.

A greater number of samples, paired with a low culling threshold, can significantly reduce the “banding” artifacts left over from wall culling. The impostor in fig 19 was rendered with
500 linear and 150 binary search steps and a cull threshold of 0.004, in contrast with that in fig 17, which used 50 linear and 15 binary steps.

Fig 19. ODR impostor with 500 linear steps, 150 binary steps
And a wall cull threshold of 0.004
6 Conclusions

6.1 Project Achievements
In this examination of impostors and GPU-based particle systems, a range of ways in which graphics hardware can be used to render systems of detailed-looking particles in real time have been discussed and key strengths and weaknesses of each method have been revealed.

6.2 Further Development
The introduction of the compute shader (Segel 2012) in OpenGL 4.3 provides an alternative to transform feedback. A compute shader would remove the need for the render operation that transform feedback brings. Compute shaders are already being considered as an option for particle systems (Bailey 2012), but at the time of writing, there do not seem too many comparisons between compute- and transform feedback-based implementations.

Similarly, a CUDA implementation of particle simulation based on the particles demo in the CUDA SDK (Green 2010) might be an improvement on a transform feedback-based implementation, and would likely provide more insight into how algorithms can be optimized for GPU architecture.

The quality and efficiency of dynamic impostors could be improved by rendering impostor textures of varying size and angles. For example, an implementation might reserve one or two low-resolution textures for small, distant and/or fast moving particles; while a larger number of larger textures oriented in a rough hemisphere around the camera, so that slower-moving, larger particles will appear correctly for the scene's perspective, and at a greater level of detail.

Rendering quality of omnidirectional relief impostors could be improved through the use of some of the improvements suggested by Andújar et al. (see “Omnidirectional Relief Impostors” in section 2) For example, taking an information-content-based approach during texture generation would improve the rendering quality of the parts of the mesh with greater detail, and would incur no runtime cost, as it would occur outside of the application.

The ODR impostors could also be improved by implementing one of the pixel-selection techniques, such as the one which prioritizes the impostor with the direction closest to the view vector. Andújar et al.’s example cannot be directly ported to this project’s implementation, as we cannot call glDepthRange from the geometry shader where the quads themselves are created, but modifying the quad’s pixel depth in the fragment shader might give a similar effect.

The solution that uses the impostor’s relief map depth is another option, and might fit better into a shader-based implementation.

6.3 Personal Reflection
This project has provided an insight into a very interesting field of graphics, and has been very informative as an investigation into GPU architecture and the associated methodology.
References and Bibliography


Charalampos, K, 2008, Granular Material Simulation, MSc Dissertation, University of Hull


Granular Material Simulation

GLM Website, http://glm.g-truc.net/ [Accessed 12 September 2012]
8 Appendices

8.1 Initial Specification and Design

3.0 Expanded specification

This project aims to develop an application that can render granular particles in a 3D scene using a choice of billboards, imposters, and true imposters. The particles will be physically simulated through Newton’s laws of motion, and will be able to collide with the terrain. The rest of the scene will be sufficiently detailed for the graphical quality to be consistent, and to show that the techniques can be implemented to work efficiently in relatively busy scenes. To this end, techniques such as normal mapping and shadows will be implemented. The scene will include height-field terrain and basic mesh objects.

The application will be able to read configuration files which will specify scene details such as gravity, the height map used for the terrain and the characteristics of the lights. In addition, it will contain the number and positions of particle systems; specifying for each system the number of particles, the texture and relief map used, size, lifetime, as well as physical attributes such as initial velocity.

The executable itself will run on the CPU, with most of the expensive run-time work taking place on shaders on the GPU.

3.1 Requirements

3.1.1 Hardware

The project will be developed on 64-bit machines with dedicated GPUs running Windows 7, though it is intended to run on 32-bit computers as well. A priority will be to regularly test the application on both AMD and nVidia GPUs.

3.1.2 Languages

C++ will be used for the CPU portion of the application, which will handle the initialisation of the shaders and the graphical objects, including particle systems. C++ is the language in which the vast majority of computer graphics code samples are written, and is reasonably fast, lacking the virtual machine overhead of languages such as Java.

The OpenGL API will be used for graphics. It was chosen as it is the developer is familiar with it, and because it is used in most graphics tutorials and code samples. The version used will be 3.0 or above, as this version brings most of the required functionality. The main alternative, Direct3D, would require additional time to learn.

GLSL will be the language of the shaders. OpenGL can compile both GLSL and Cg, but as the latter requires the extra step of integrating nVidia libraries, the former was chosen. Version 1.3 or above will be used, to coincide with the chosen OpenGL version.

3.1.3 Software and Libraries
FreeGLUT will be used to handle the application's windows and the loading of OpenGL functions and context. It was chosen at it is lightweight, easy to use, and up-to-date. It is on the X-Consortium licence, allowing it to be freely used. An alternative to FreeGLUT is FLTK, a cross-platform GUI toolkit with built-in GLUT emulation. FLTK was not chosen as much of its functionality is unneeded.

The OpenGL Extension Wrangler (GLEW) will be used to handle the loading of functions and extensions. Like FreeGLUT, it is lightweight and easy to use. It is on the MIT licence, allowing it to be freely used. An alternative is Glee (OpenGL Easy Extension Library).

Boost libraries might be used for additional functionality such as timing, or data structures such as hash maps. The majority of Boost libraries are released under the Boost Software licence, which allows them to be freely used.

GLM might be used to implement mathematical functionality. GLM, a vector and matrix mathematics library designed to share its syntax with GLSL, might be used for mathematical functionality. GLM is available under the MIT licence.

Blender and GIMP are likely to be used for the creation of models, textures and normal maps.

3.2.7 Initial Class Design

8.2 castRay Function for Omnidirectional Impostor Particles

```cpp
cvec2 castRay(in sampler2DArray rm, in vec2 tc, in vec2 delta, out float bestDepthOut,
              out float bestDepthIn) {
    const int nLinearSteps = 50;
```
const int nBinarySteps = 15;
float rayDepth = 0.0;
bestDepthOut = 0.0;
bestDepthIn = 1.0;
float stepSize = 1.0/float(nLinearSteps);
float texelDepth = texture(rm, vec3(tc, fID)).a;
float intersect = 0.0;

// linear test
for(int i = 1; i < (nLinearSteps + 1); i++) {
    intersect = 1.0;
    if(texelDepth >= rayDepth) {
        rayDepth += stepSize;
        texelDepth = texture(rm, vec3(tc + (delta * rayDepth), fID)).a;
        bestDepthOut = texelDepth;
        intersect = 0.0;
    }
}
bestDepthIn = texelDepth;
bestDepthOut = texelDepth - stepSize;

// "Rewind" to the point before the intersection
rayDepth -= stepSize;

// binary search
for(int i = 0; i < nBinarySteps; i++) {
    stepSize *= 0.5;
    rayDepth += stepSize;
    texelDepth = texture(rm, vec3(tc + (delta * rayDepth), fID)).a;
    if(texelDepth <= rayDepth) {
        rayDepth -= stepSize * 2.0;
        bestDepthIn = texelDepth;
    } else {
        bestDepthOut = texelDepth;
    }
}

return (tc + (delta * bestDepthIn * intersect));