I would like to dedicate this work firstly to my parents, who have offered me unwavering support throughout my life. Without their guidance, support and discipline (particularly during my formative years) I would not be who or where I am today. Secondly to my sister, who has managed to put up with me thus far without going insane. She has acted as a role-model, a tutor, a parent, a friend and a house-mate. My love of sci-fi and general nerdiness can largely be attributed to her.

Finally to the three main tutors in my university career: Minsi Chen, Tommy Thompson and Wayne Rippin. Each has taught me not only in an academic sense, but also in a personal one. I truly believe that without their support, patience and understanding, I would not have made it this far.

Thank you all.
Abstract

Traditional rendering methods such as Rasterisation, Ray Tracing and Deferred Rendering have proven to be suitable when rendering scenes with a high level of visual fidelity. However, they have proven to be problematic when it comes to certain types of materials and objects including deformable meshes. In this paper, we present a number of techniques based on a Point Based Rendering pipeline, and analyse the results against a traditional forward pipeline.
3.3.7 Engine Performance Analysis ........................................ 24
3.4 Experimentation .................................................. 24

4 Results and Analysis ............................................. 26
  4.1 Engine Performance ............................................ 26
    4.1.1 Subdivision .............................................. 26
    4.1.2 Back Face Culling ....................................... 27
    4.1.3 Object Splitting ........................................ 29
  4.2 Performance of Splatting Techniques ......................... 29
  4.3 Comparative Performance Analysis ............................ 33
  4.4 Visual Quality ................................................ 33
    4.4.1 Polygonal Rendering vs. Point Cloud .................... 34
    4.4.2 Comparing the Quality of the Splatting Techniques .... 35

5 Conclusions and Future Work .................................... 38
  5.1 Findings and Reflection ...................................... 38
  5.2 Known Issues and Improvements ............................... 39
  5.3 Final Thoughts ............................................... 40

6 Bibliography .................................................... 41

Appendices ......................................................... 42
A
  Machine Specification ........................................ 42
B
  Control Scheme ............................................... 43
C
  Geometry Shader ............................................. 43
D
  Back Face Culling Algorithm ............................... 47
E
Subdivision 49

F
Decimation 52

G
Object Splitting 53

H
Engine Class Diagram 55

I
Spider-Man Subdivision Iterations Using our Three Splatting Techni ques 56
Introduction

Since the dawn of video games graphics, both hardware and rendering techniques have strived for both higher visual fidelity as well as efficiency. The goal is to visually represent vast, versatile worlds in a three-dimensional environment on a raster-based display. However, before we can explore the common techniques used for graphical representation of models, we must first explore what defines a model.

1.1 What Makes a Model?

Models are usually represented by polygonal meshes that are often constructed by an artist, 3D modeller or sculpting program such as ZBrush. Typically, the vertices used to describe the points in 3-Dimensional space hold all the information needed for rendering, including their position, normals, colour and texture coordinates. Whether the polygons constructed from the vertices are triangles, quads or any other geometry does not matter, as long as the vertices have all the information needed. Not all Rendering techniques need this however. Ray Tracing, a technique we will explore in detail later on, uses the idea of a ray (or light photon) to find when light would hit on an object. The object need not be described, instead a mathematical description (perhaps of a sphere or cube) can be used. However, this still requires vertices to hold the information needed to represent the information, since the fired ray returns all the information in terms of a vertex.

This reveals a clear correlation between two main forms of rendering, Rasterisation and Ray Tracing. Both require vertices to hold all the information needed to render objects, whether they are in the form of a mesh or not. With this information, techniques based around using the vertices themselves, without any other knowledge, have been developed. These are collectively known as 'Point Based Rendering' techniques, and are becoming increasingly popular due to the development of laser scanner technology that can be used to easily generate a collection of vertices that represent a real world object.

1.2 The Rasterisation Pipeline

The Rasterisation pipeline is used to convert the three-dimensional object into a raster state, that can then be represented on a two-dimensional device. This is often completed
by objects known as shaders, which allow the GPU to compute the position and colour before rendering.

The input and geometry generation stage centers around reading in the raw model data. In many engines, the data is parsed and stored in custom data types representing triangles, quads or polygons, based on how the mesh was serialised. It is also at this stage that any missing data that might be needed (e.g. normals, bi-normals and tangents) are calculated and stored. Once we have the data stored, we are able to start the transformation process, bringing the vertices from model space to viewport space so that they can be rasterised. This is done through a series of matrix multiplications, resulting in a change in coordinate system within the vertex. Following the transformation into world coordinates, we need to transform the vertex into view and viewport spaces.

At this point, there is often a lot of polygons ready to rendered that would not be seen. Back face culling is a technique used to reduce the number of polygons that need to be rendered based on this principle. On average, this technique can reduce the number of rendered polygons by around 50%. The technique involves using dot product to calculate the difference between each polygons normal and the view vector, then culling the face if the polygon’s normal is in line with the view and not opposing. Even though we have culled 50% of the polygons for each mesh, multiple meshes can still cause redundant rasterisation. This can happen when, from the view of the camera, multiple models overlap. As a result of the overlap, multiple polygons that will not actually be seen will still be rendered, costing valuable performance. However, if the object at the front is in any way transparent, then depth testing should never be done, as polygons from the object behind will be culled when they should be visible. Depth testing revolves around organising the Z-values of polygons so that we can easily identify what will be at the front. We can then write these to the back buffer in a back-to-front fashion. This way, polygons that are behind others will be overwritten, causing them to appear behind the closest object to the camera.

The pixel shader is the final stage that writes to the back-buffer. The shader takes the information for each polygon and shades the individual pixels (or fragments) based
on an infinite number of effects, including lighting, texturing, normal maps, emission maps and shadows. This can produce almost any effect imaginable. The shader uses scanline conversion and interpolation to decide the final fragment colour. Scanline conversion is the process of deciding whether a pixel should be shaded. This is based on the edges of the polygon, filling the pixels between the edges that have been passed to the shader. The other technique used, interpolation, is a common technique used to find a single point between two points. In shaders, this is often used to create dynamic effect by interpolating between colours, position, texture coordinates and normals.

1.3 The Ray Tracing Pipeline

Rasterisation is not the only technique used to represent realistic environments. Ray tracing is used to create some of the most realistic looking scenes that are rendered. The concept revolves around mimicking light photons by firing a ‘ray’ from the camera at the scene, and following its’ bounces to accurately attribute all of the lighting and colouring from the surrounding area.

Figure 1: An image of a scene rendered using ray tracing (Millward, 2015).

Unfortunately ray tracing requires a lot of computation, and often struggles to be run in real time. This means that the technique has never solely been used in games or other real-time applications, with the current limitation being on the hardware. Fortunately,
the complexity can be reduced at the cost of visual fidelity by reducing the number of
bounces the ray is followed for, but not enough to present a real-time technique that is
of a high enough quality to compete with rasterisation based techniques.

Although there may be many drawbacks of ray tracing, there is one particular advantage
that we would like to focus on. Due to the nature of the technique, polygons are not
necessarily needed. Instead, the ray can be tested against the point data to determine
if a ray would hit. This presents the idea of points being used for uses other than for	polygon construction.

1.4 Point Based Rendering

Points are the most critical part of either basic pipelines. In both cases, points can be
used to construct polygons, providing texturing, normal and positional information to
the pipeline. However, this is not the limit of the points’ usefulness. As they hold all the
information, we can dispense with polygons all together, using the points themselves the
aid in the rendering of objects. This is the basis for the technique known as ‘Point Based
Rendering’.

Point Based Rendering became particularly useful with the development of laser scanners,
which can scan real world objects and produce a point cloud representing the scanned
object. The technique revolves around representing the points themselves, instead of
building polygons. Each point is covered with a disk, and all the disks are then blended
to create the desired surface, this is known as splatting. This technique is often more
efficient that polygonal mesh rendering, but at the cost of visual fidelity.

Point Based Rendering is particularly suitable for deformable objects, a concept that has
been difficult to represent using traditional methods. Deformable objects are typically
rendered using meshes, which are modified in real-time at great computation expense.
This is not ideal, as the consistency of the mesh is in question Majumdar, n.d. However,
since the points are used to represent the hull of the object, the points themselves can be
moved without care for the attached polygons or the effect of consistency.
1.5 Goals and Objectives

We aim to develop a point based rendering pipeline based on the current state of the art. In order to achieve this, we have the following objectives:

- Discover the current state of the art.
- Determine if point based rendering is more efficient than traditional rendering methods.
- Determine if using point based rendering results in a loss of quality in comparison to traditional rendering methods.

In this report, we’ll discuss the history of related computer graphics, as well as the potential benefits of Point Based Rendering. We’ll then discuss how we intend to implement our technique and develop our engine in order to gain the results we need, before moving on to analyse the results and provide our conclusions. In order to determine the success of our engine, we will compare and evaluate both the computational times and the visual quality of both traditional rendering methods and point based rendering while rendering various objects in real time. We will also analyse the impact on hardware, using GPU analysis to discover the time taken on the GPU to render.
2 Literature Review

2.1 Point Based Rendering

Point Based Rendering alone attempts to address the failings of the previously mentioned methods by removing the polygons. Instead, sets of unorganised points, gathered using 3D laser scanners, are used as the base information. As the points are unorganised and hold no information of connectivity, any point-based rendering implementation needs to sort the unorganised points before they can be used (Mller et al., 2004).

Once this is done highly detailed objects can be rendered as well as irregular shapes. Point based rendering is more suited to these types of objects due to the ability to match the used resolution, along with the ability to re-sample as needed (Zheng et al., 2002). This technique is not suited to rendering objects which are simple and smooth, a wall for example, due to the computational complexity involved. Traditional methods, such as polygonal meshes, are far more efficient at rendering these low-detail objects. Little research into the propriety of point-based rendering for deformable objects has been completed, although its ability to render interactive water has been explored.

We will now explore point based rendering, along with previous works related to deformable objects, in depth.

2.1.1 Points as a Primitive

The main benefits of the point-based rendering technique stem from the use of points as a primitive as opposed to geometry such as triangles or quads. Levoy et al. identify three problem areas with traditional methods that can be solved by using points as a primitive (Levoy and Whitted, 1985). These are:

- Customised rendering algorithms, one for modelling and another for displaying;
- Rendering order (object vs image);
- Object complexity vs it’s coherence. This is the trade-off between quality and efficiency.
Levoy and Whitted propose using zero-dimensional points as a meta-primitive, addressing each problem area in turn. As points are universal, there is no need for custom algorithms to be made, as the points can be displaced in space and still rendered as a continuous three-dimensional surface. As there is no connectivity or coherence between the points, they can be rendered in any order and still produce the same effect.

2.1.2 Surface Reconstruction and Surfels

The first step in any point based rendering technique is to acquire a set of points. These are easily gathered using accurate 3D laser scanners. Unfortunately, these point sets are often not organised; this leaves us with the task or organisation. One technique commonly used to deal with large point clouds is to generate a triangulated mesh from the point data, and then use polygon reduction techniques to reduce the number of polygons (Zwicker et al., 2001). These are usually developed on a case-by-case basis, resulting in redundancy and a loss of time. With this in mind, Hoppe et al. produced a generic algorithm for surface reconstruction that can be used on both contoured and non-contoured point sets (Hoppe et al., 1992).

Point based rendering allows us to use the raw data instead of constructing a polygonal mesh first, bypassing the need for this all together. To do this, we use a technique called segmentation to separate groups of points based on common factors (Vano, Hamann, and Brunnett, 2008). This is usually completed using a neighbourhood graph to estimate knowledge (such as normal vectors) about the points (Vano, Hamann, and Brunnett, 2008). Vančo et al. propose multiple algorithms that can discern meaningful quadratic segmentations from unorganised point sets (Vano, Hamann, and Brunnett, 2008). These segmentation algorithms are based on various properties and the quality of the scan. A low quality scan can make this incredibly difficult, along with noise and non-uniform sampling. Once organised, we move on to splatting them.

Splatting is a method of first calculating which fragments need to be shaded, and then shading them using a shaped surface element, called a surfel (Pfister et al., 2000). These are weighted elliptical (or rectangular) disks that cover the points, giving an appearance similar to that of a triangulated mesh, but at a significantly lower computational cost.
Using surfels gives us the ability to remove hidden surfaces, provides edge anti-aliasing and order-dependent transparency (Zwicker et al., 2001). They are highly suited to organic models, with high detail, but are ill suited to large and flat surfaces such as walls. Surfels allow the bulk of computational time to be moved into the preprocessing stage, causing a significant increase in efficiency vs. traditional polygonal techniques (Pfister et al., 2000).

2.1.3 Deformable Objects and Point Based Rendering

Creating interactively deformable objects has been a large focus of study in the past. In 2006, Christian Duriez et al. described a method of using the laws of friction and laws of contact to describe the change to points needed for haptic feedback for deformable objects (Duriez et al., 2006). The resulting algorithm is not overly fast, but the authors acknowledge that it is not optimised and has room for efficiency improvements.

Deformable objects are key to computer graphics because very few things in reality can be modelled as rigid bodies. The most obvious is a human. As an entity, we are not rigid, humans are deformable. However, despite their importance, achieving accurate real-time deformable objects has eluded us. Most approaches thus far have sacrificed either accuracy or speed. ‘Interactive models’ aim for speed, and sacrifice the accuracy of the simulation. Typical examples of this are mass-spring models. ‘Accurate models’ do the opposite. They aim for accuracy, while sacrificing speed. These tend to be offline due to how computationally expensive they are (James and Pai, 1999).

James et al. present a new technique that takes the best of both worlds, producing an efficient model that provides accuracy without the sacrifice of speed. This is done through a combination of precomputed system responses and a limited amount of non-zero boundary data associated with typical interactions (James and Pai, 1999).

Developing realistic-looking water is a particularly difficult task in the world of computer graphics. The most common method used is that of a particle system, using many small particles (geometric models, most commonly spheres) to represent the water. Unfortunately, these are inefficient and prone to gaps. Using point based rendering, along
with a dynamic surface sampling density, close-fitting particles can be used to accurately visualise deformable effects (Mller et al., 2004). Iwasaki et al. propose an efficient algorithm that not only uses point-based rendering for water, but also handles splashes effectively at a high rate (Iwasaki et al., 2006). While they focused on achieving realistic splashing, their framework was unfortunately run entirely on the CPU, and only produced clear, shallow water.

Although water is the prime focus of this paper, there are many other deformable objects that exhibit translucent properties that benefit from point based rendering, and some previous work has been completed examining this. The visual effects of materials such as skin, milk and wax, are caused by subsurface scattering, the diffusion of the incident light once it has penetrated the material. Work on this has been done for static objects, and offline rendering. However, Mertens et al. have shown that this can be produced in real-time on deformable objects (Mertens et al., 2003).

2.2 Newer Forms of Rendering

2.2.1 Deferred Rendering

Deferred Rendering focuses on efficiency, attempting to reduce the number of times each fragment is rendered. In traditional methods, if a fragment receives light from more than one source, the fragment is rendered once for each source in order to calculate the final colour additively. This requires re-submitting the geometry for each lighting pass required.

However, deferred rendering renders all the light information to a single two-dimensional texture, and then samples the texture once to determine the final colour for rendering. Multiple light passes do not require the geometry to be resubmitted. Instead, the geometry data is stored in a buffer that is exposed to the shaders, allowing them to be sampled multiple times from the same source.

This is typically done for each light, as well as shadows, bump maps, depth and emissions. Once all the textures are generated they are then sampled during a final pass, adding all
the information together to get the final colour of the fragment before writing it. This massively improves performance as calculations are only performed once, and writing the fragment is only done once.

Deferred rendering has a drawback in terms of implementation though, as none of the calculations implemented in a standard pipeline are usable, save the positioning of polygons (Deering et al., 1988). The change from submitting the geometry once and rendering each fragment sequentially to storing the geometry and rendering multiple times while sampling and generating textures is the main cause, requiring most pipelines to be custom written in order to handle the changes deferred rendering presents.

2.2.2 Forward+

Forward+ rendering attempts to do a similar lighting optimisation as Deferred Rendering but on a smaller scale. The world is described as a 3-dimensional grid composed of ‘tiles’. Each tile has its lighting calculated together, meaning that instead of each fragment needing to be rendered multiple times, the data for the light it receives is already computed for the tile the fragment resides in (Thatcher, 2014).

This technique is often coupled with other pipelines, such as deferred rendering or rasterisation, as it performs a typically expensive procedure into a more efficient process. It’s important to note that Forward+ is not in itself a pipeline, but is instead an efficient tool to be incorporated into others. Unfortunately it is not without flaws as the technique can easily lead to false positives, meaning fragments can be incorrectly lit.

These false positives most often occur when both foreground and background elements are present in a single tile. Harada proposed a 2.5D culling approach, which splits the viewing frustum along the depth direction (Harada, 2012). This is then stored as a depth mask, and sampled for each tile check, reducing the number of overlapping lights per tile. Harada reported that in their test scene without their algorithm there were many tiles with over 200 light overlaps. However, when their algorithm was implemented, the most a single tile had was 120 light overlaps. This clearly demonstrates the benefit of the algorithm, but more work is needed to explore the extent of the benefits.
2.3 How This Impacts Our Implementation Goals

Even though Forward+ and Deferred Rendering have many benefits over a traditional forward pipeline, we’ve decided to use the forward pipeline in order to make the analysis and comparison easier between point based and polygonal mesh based rendering. However, we feel it is important to address the benefits that the two rendering techniques could pose to point based rendering.

Deferred rendering, as we have discussed, focuses on reducing the number of times each fragment is rendered. This technique is targeted at the lighting information, but the texturing and normal mapping can also be improved through the same technique. However, as we are focussed on the quality and consistency of the model produced, not the lighting and texturing of the model, we believed it would be unnecessary for us to implement this technique. Forward+ is also focussed on improving the lighting calculations. With the scene divided into a grid, each tile is lit separately. This would be very beneficial if we were aiming for a dense scene with over 1000 lights, but as we are not it is again not necessary. However, the grid system could be used to determine where more points are needed, with each tile being subdivided based on the number of points present in the tile.

With all of this in mind, we are planning to implement an array of features into the engine in order to aid the development and analysis of the point based rendering technique we aim to develop. For the point based rendering itself, we intend to use the geometry shader to build new geometrical shapes for splatting. We also intend to use normal mapping and displacement mapping to represent water, and to determine where points should be moved to.
3 Methodology

The engine set-up we chose to implement is a forward pipeline that has some tailoring toward point based rendering. This allows us to easily switch between polygonal meshes in the forward pipeline and the point clouds being rendered, to aid in our development and analysis.

3.1 Our Pipeline Vs. the Forward Pipeline

As previously described, a typical forward pipeline will utilise shader programs to rasterise a polygonal mesh, allowing for 3-Dimensional objects to be viewed on 2-Dimensional displays. We decided to start with a fully implemented forward pipeline, allowing us to render polygonal meshes which we intend to use as part of our rendering quality analysis.

Once the forward pipeline was in place, we needed to add a second layer to the pipeline allowing point clouds to be rendered as well as meshes. The main task in this process was to design the splatting implementation, finally deciding to use the geometry shader to generate the splats. We designed three splatting techniques, points, disks and rectangles, which each require new vertices to be generated from the geometry shader.

Although we added this second layer to the code base that handles the point clouds, our implementation is still fundamentally based on the forward pipeline. As a result of this, we require connectivity information for the vertices, meaning raw point clouds don’t work in our engine. Instead, polygonal meshes are still required for our engine to operate.

In a typical point based renderer pipeline, the surfels that are generated would be fitted based on an orientation parameter. Often, this parameter is based on the normal of source point, and the surfel rotated so that it fits into place.
Figure 2: A typical approach to splatting (Pfister et al., 2000).

Our implementation is less than typical. Instead, we splat in image space, resulting in all our surfels being orientated toward the camera. This approach can be seen in figure 3a. As a result of this technique, the far edges of any object being rendered will have an aliased effect. This is shown on a rendered Stanford Bunny in figure 3b.
(a) A representation of our splatting technique.

(b) Aliased result of our splatting technique.
3.2 Point Based Rendering Implementation

3.2.1 Geometry Shader

In our implementation (which can be viewed in appendix C), the geometry shader does a large amount of the work, allowing us to make use of the GPU’s power. The geometry shader, which sits between the vertex and pixel shaders in the pipeline, takes each point in and sends it through to the pixel shader as is. This allows us to turn the polygons into points very easily.

The geometry shader also has implementations for various shapes and sizes of surfels. The new geometry for the surfel is constructed in the shader, before being sent to the pixel shader for rasterisation. At the same time, we use interpolation schemes to determine the texture coordinates of the new vertices being generated. This is much more efficient than constructing the geometry on the CPU, and results in faster load and rendering times. However, the size and shape of the surfel has an impact on performance vs other forms of surfels.

We also decided to not to orient the surfels themselves, and instead treat them as billboards which always face the camera. This decision was made as a prioritisation, noting that orienting the surfel based on its normal is relatively trivial, and thus decided to focus on different features.

3.2.2 Culling

Due to the nature of the pipeline, techniques such as Z-Buffering and Depth testing don’t work as they rely on polygons. This is also true for API implemented culling methods. With this in mind, we implemented our own software side implementation of back face culling, testing the polygon’s normal before it is processed by the geometry shader and modified to points. This allows us to reduce the polygon count and subsequently the point count by roughly 50%.
3.2.3 Subdivision and Decimation

Subdivision is the act of splitting a polygon based on set criteria. To help with the visualisation of the points, and given that most of our test models were not sufficiently dense with points, real-time subdivision has been implemented (appendix E). In our implementation, each triangle is split into four smaller triangles by generating three new points along the edges, resulting in three new points per triangle per subdivision. This is particularly useful to see how many times objects need to be subdivided in order to get enough points to fill the image.

![Figure 4: Conceptualised subdivision.](image)

Decimation is the reverse of this process. We take a collection of four polygons that originally made up a larger polygon and remove the new points, creating a larger polygon. This can be used to reduce the polygon count, but does result in a loss of visual fidelity as the poly count is reduced. This was primarily implemented into the engine as a way of visualising the point data, and to allow real-time manipulation of the point data. The algorithm can be seen in appendix F.

3.2.4 Displacement Mapping

Height maps are two-dimensional textures used to represent the height of a particular location on geometry. These are used to displace the points, using the texture coordinates to locate each points relative position on the height map, and then shift it’s Y value based on the height represented. This can be used to displace the points in order to represent waves in an ocean. When combined with a parallax effect, this can be used to model a representation of wave motions. However, this is not realistic as the waves would be
deterministic in their nature, being based on a static two-dimensional texture.

![Image](image.png)

Figure 5: Displacement Mapping on a Point Cloud of a Plane.

### 3.2.5 Visual Effects

Along with the displacement mapping, which aims to manipulate the geometry in order to represent water, we use a range of visual effects in the shaders to achieve a more believable water look. This includes normal mapping, which is used to modify the normals of a particular point to show subtle detail and give an illusion of depth, texturing and lighting. These effects are all managed through the vertex and pixel shaders, allowing the effects to be modified either offline or in real time as the engine is running.

### 3.3 Engine Design - Ease of Use

#### 3.3.1 Hot Compiling Shaders

In order to aid in the development of not only the implementation of Point Based Rendering, but also to add versatility to the engine itself, functionality for hot compiling of shaders was added. This allows for the shaders to be modified and updated while the engine is running. This is particularly useful when modifying lighting or texturing settings to achieve a desired effect.
If a shader does not compile correctly, the engine defaults to the previous state of the shader, preventing an engine crash or unfavourable side effects. However, once the engine has been shut down, the shader will be in the modified state, meaning it will not allow the engine to re-launched until the error has been corrected.

### 3.3.2 Use of Template Functions

In order to facilitate the easy switching between meshes and point clouds, as well as improving the flexibility of the engine, Template functions were implemented. These allow for types to be passed into the function, and the type then used to instantiate the objects. As the ‘Point Cloud’ and ‘Mesh’ types are both inherited from a ‘Renderable’ object, both can be treat as the same despite being separate types.

### 3.3.3 Camera and Light Movement

As with most engines, we have supported camera movement through the scene in a ‘Ghost Mode’. This means there is no collision on the camera itself, allowing the camera to view any object from any angle. In order to aid the rendering and testing of various systems, the light has a similar control system, allowing it to be moved to any position in the scene. There has also been a ‘reset’ implemented, allowing the user to reset the camera and light position to how it was when the engine was started.

### 3.3.4 Dynamic Game Object Creation and Scene Modification

To speed up the setting up of scenes, the idea of a ‘Game Object’ was encapsulated. This means that new objects can easily be added simply by defining them and adding them to the objects list. To improve this further, we added serialisation so that the scene can be described in a text file which is then read into the engine. This allows for quick scene set-up prior to running the engine. A hot-key was then set up to reload the scene from the text file, so modifications can be made once the engine is running and then the scene can be loaded at the touch of a button.

In engine:

```cpp
PointCloud* obj = new PointCloud(mat, bump, height, tempVerts,
                                    tempIndices, scale);
objects.push_back(obj);
```
In the text file used for scene descriptions:

plane2.obj:water.png:waterBump.png:waterHeight.bmp:0.5

3.3.5 Toggle Rendering Method

Due to the nature of this report, we needed to easily compare multiple rendering methods. The easiest way to do this was to have a button that toggled between rendering methods, but kept all other information (models, scene data, camera and lights positions etc.) the same. Currently this button allows the user to toggle between traditional Mesh rendering and our implementation of Point Based rendering.

3.3.6 All Axis Rotation

Each individual game object has the ability to be rotated on each of the X, Y and Z axis either individually or simultaneously. This allows the objects to be seen from various view points easily, especially when coupled with the camera movement. The rotation speed can also be modified in real time, and the rotation direction can be reversed by reducing the rotation speed to negative numbers.

3.3.7 Engine Performance Analysis

The engine provides real-time feedback on its performance as well as helpful information for the user. This includes frame rates, a point count, the number of subdivision iterations, the camera position and the light position. This data is updated every frame with the exception of the frames per second count, which is updated every second.

3.4 Experimentation

To adequately measure the success of our implementation, as well as the engine as a whole, a series of experiments have been devised which can be split into two categories. First, the engines performance, with each techniques impact being measured through the change to the average frame time. A large sample will be gathered, and then averaged in order to gain a reliable result.
Alternatively, we may measure the success of a particular implementation by the results of the process. Subdivision, for example, will negatively affect the frame rate. However, the frame rate is second to the points being generated, so we will measure the points themselves. For all of these results, we have decided to use a model commonly used for such result gathering, the Stanford Bunny. This a suitably dense point cloud model allowing us to thoroughly explore the suitability of both our engine and our implementation.

Secondly, we need to measure the quality of the rendering produced by our implementation. As quality is a qualitative measurement, this is harder to analyse. Again, we intend on using the Stanford Bunny, however we will also use a textured Spider-Man model to analyse the quality of our texture coordinate calculations, as well as analyse the splatting techniques on a more complex model. We are also using these models because we wish to analyse the relationship between how dense the original point cloud is vs. the quality produced. The Stanford Bunny is a very dense model, where the Spider-Man has both dense and sparse areas.
4 Results and Analysis

As previously discussed, we intend to measure the performance and success of our engine generally, as well as our implementation specifically. This will allow us to gain an overall understanding of how successful we have been. In the following sections, we will dissect the engine and the implementation and measure their adequacy and their impact on performance.

4.1 Engine Performance

In order to quantitatively analyse the performance of our engine, separate features that impact performance are measured individually. These features and their impact are separated and analysed in the following sections, along with a critique of the implementation itself where appropriate.

4.1.1 Subdivision

Each iteration of the subdivision algorithm multiplies the number of vertices. The numbers of points can be viewed below for different objects for each iteration. The results of this are represented using the Stanford Bunny model in figure 10.

<table>
<thead>
<tr>
<th>Model</th>
<th>Base Points</th>
<th>After First Iteration</th>
<th>After Second Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bunny</td>
<td>34834</td>
<td>416712</td>
<td>1666848</td>
</tr>
<tr>
<td>Spider-Man</td>
<td>6262</td>
<td>69918</td>
<td>279672</td>
</tr>
<tr>
<td>Plane</td>
<td>1089</td>
<td>12294</td>
<td>49176</td>
</tr>
</tbody>
</table>

As you can see, the number of points exponentially grows with each subdivision iteration. This is not ideal, and actually represents a flaw in the algorithm we devised for subdividing. The issue stems for the addition of points to the collection, ignoring the possibility that the point already exists. This causes excess rendering and processing power that could be avoided.
4.1.2 Back Face Culling

In an effort to improve efficiency, a back face culling algorithm was implemented that attempts to reduce the point count by fifty percent. This suffered from the same issue that our subdivision algorithm did, adding points regardless of whether they were already present. This caused the point count to increase by around three hundred percent, instead of decreasing by fifty percent. As this entirely defeated the point of the algorithm, we implemented a method for detecting whether the point that is being added already exists in the container (which can be seen in its entirety in appendix D). However, this had the undesirable side effect of increasing the load-time, as the container of vertices needed to be searched for each new point that could potentially be added. This solved the problem and resulted in the desired fifty percent decrease. We also only run the algorithm once, on load, in an effort to reduce the impact on the performance. This does reveal itself when the object is rotated though.
Number of points generated after each iteration of the subdivision algorithm

![Graph showing number of points generated after each iteration of the subdivision algorithm.](image)

**Figure 7:** The Stanford Bunny rotated to reveal the effect of Back Face Culling
4.1.3 Object Splitting

While investigating an issue, an interesting discovery was made. An algorithm (appendix G) was devised for splitting a loaded object into several smaller objects based on the number of points the original consists of. This resulted in a significant increase in performance. However, this also resulted in a rendering issue, causing a depth issue due to the objects rendering order.

As you can see, there is a clear benefit to splitting the object into many smaller objects, and rendering them individually. The graph shows a linear correlation between the performance and the number of sub-objects we generate. This linearity means we can accurately predict the result of further object splitting, with a small margin of error. The performance gain itself can be explained by the improved bandwidth efficiency. Smaller buffers are a lot quicker to both iterate through and process in the pipeline, resulting in the decreased frame time. However, the cost of the rendering quality has yet to be solved, resulting in the algorithm for splitting the object being only useful when testing certain features, such as animations and game object manipulation.

4.2 Performance of Splatting Techniques

In an attempt to discover the most optimal splatting technique, along with the technique that presents the best visual quality, several different geometry based splats were
implemented (which can all be seen in the geometry shader in appendix C). These were generated through the geometry shader, which is present between the vertex and pixel shaders in the pipeline. A set number of vertices are passed forward based on what is needed, although this is usually either a single vertex or the three that construct a polygon. We implemented three separate techniques, which each effected the performance in different ways due to the volume of the geometry being dynamically created by the geometry shader.

Discs were the hardest geometric shape to produce, and had the highest impact on the performance of the engine when chosen. This is due to needing to dynamically create the polygons that construct the disc. Each disc is a two-dimensional filled circle which is actually a billboard, rotated toward the camera. These are built with the original point in the centre, with the surrounding points being mathematically created. Each point also has generated texturing coordinates based on the original point, using the same calculation that is used to generate the points’ position. The number of points on the circumference of the disc also effects the performance.

Impact of the number of points on the circumference of a disk on frame time

Rectangles are created on a similar principle. The four corners are generated with the original point being centred, and texturing coordinates are generated based on the same distance from the original point. As only four new points are generated, the rectangles proved to be much more efficient vs. the discs.
The most efficient of the three techniques was the points themselves, as there are no new points being dynamically created by the geometry shader. Instead, the technique relies on the subdivision to create new points, and simply renders those points themselves. This is relatively easy, given the points already hold all the information needed.

As the graph presents, the disc is the most inefficient. However, both the rectangle and the points are more efficient, with the points gaining around one-hundred and twenty frames vs. polygons. This data was gathered by rendering the Stanford Bunny model, with the disc generating sixteen points for its circumference.
Figure 8: The Bunny’s base points being rendered with different splatting techniques.
4.3 Comparative Performance Analysis

In order to gather a set of results that were consistent, I used the profiling tools in Visual Studio 2013 edition to analyse the CPU times of the four key techniques we employed. The following results were gathered from a single session, during which the following techniques were used. The results have been verified by running the same experiment five times, with near identical results.

<table>
<thead>
<tr>
<th>Technique</th>
<th>CPU Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back Face Cull</td>
<td>25,000</td>
</tr>
<tr>
<td>Split Objects</td>
<td>15,000</td>
</tr>
<tr>
<td>Subdivision</td>
<td>10,000</td>
</tr>
<tr>
<td>Decimation</td>
<td>5,000</td>
</tr>
</tbody>
</table>

These results back-up our expectations, with our back face culling algorithm being very slow thanks to the search being used. However, we were surprised to discover that splitting objects was slower than subdivision. Upon further investigation, this is reliant on not only the number of object we wish to split the original model into (our experiment involved splitting into ten objects), but also how dense the original point cloud is.

4.4 Visual Quality

One of the most useful indicators of the performance of our engine available to us is the quality of the renders produced. This is a rather subjective process, and as the creators we are biased. However, we hope to present compelling evidence and critically analyse some renders taken from the engine, and compare those to the frame rates previously discussed in an attempt to gain some insight and determine whether our implementation is valuable. This process may also help to identify key areas that can be developed on in the future, in order to improve the process.
4.4.1 Polygonal Rendering vs. Point Cloud

Although the analysis of the performance for the techniques themselves was extremely useful, it would be remiss to ignore the quality comparison of the final effect. As such, we will now analyse the visual fidelity of Polygonal Rendering vs. Point Based Rendering.

As this progression shows, subdivision on meshes doesn’t accomplish anything in terms of visual quality. This is due to the subdivision technique used, only splitting the polygon in its current form to several smaller polygons that occupy the same space.
Figure 10: Increase in quality through Subdivision while rendering Point Clouds

Unlike the mesh, subdivision has a fundamental role in the quality of the rendered model when dealing with Point Clouds. This is due to the points themselves being used to represent the surface of the model. As this progression shows, three iterations of the subdivision algorithm are needed to have the rabbit model represent it’s mesh equivalent.

4.4.2 Comparing the Quality of the Splatting Techniques

When generating the disk, rectangles and points (through subdivision) we used an interpolation scheme to calculate the texture coordinates of the new vertices. This usually involved using the same mathematical method we used to create the new vertex, and applying it to the texture coordinates of our source vertex. However, for the rectangles we used a slightly different method. This involved taking the distance between the source vertex and the new vertex, and then dividing that by the size of the texture. This gave us the distance to add related to the texture coordinates. We used a textured Spider-Man model to see the results of the texturing. Below is the model rendered using points, disks and rectangles after four subdivision iterations. The previous three levels of subdivision
can be found in appendix I.

(a) Points

(b) Disks
These images reaffirm our previously stated suspicions. Disks do not fill enough of the surface area even after four iterations. Rectangles can fill the surface area earlier in the iteration process, but the visual quality is lacking. Points still represent the best quality. Four iterations were needed to cover the surface in points, but the performance is still more than acceptable after this process.
5 Conclusions and Future Work

As with any engine, there is always room to improve. In the hope that this work will prove useful to someone else, and with the interest of helping them improve upon our work, we will go over some of the issues we have identified and any suggestions we have about how to approach them. However, first we would like to address the original objectives we laid out in section 1.1, and reflect upon them now that the project has come to a close.

5.1 Findings and Reflection

As we discovered as the project progressed, point based rendering is a very reliable and flexible technique. The performance hit of the tested splatting techniques was undesirable, with disks providing no benefits to offset the drawback of this. On the other hand, rectangles posed a minimal performance hit and presented a possibility for being very suitable for splatting. In our implementation, the rectangles were not oriented, simply billboarded in view space. However, if the rectangles were oriented and blended toward to edges, rectangles could pose a viable solution to the splatting problem.

The results of the rectangles indicate that low vertex geometry may make good surfels. With this in mind, it would be beneficial to explore making use of hexagonal geometry as surfels. Hexagons are a particularly suitable geometrical shape due to a number of factors. Firstly, the distance between the center of a hexagon and its’ neighbours will always be equal. Secondly, hexagons that are touching will always share only a single edge. These two factors would prove very beneficial for splatting, particularly if the technique was performed in image space, much like our rectangle technique.

Finally, we believe the geometry shader is more of an appendage than a body itself. To clarify, the geometry shader is perfect for taking polygons and just rendering out the points, there-by cutting down on a lot of the rendering and processing power needed. However, this relies on the subdivision having already taken place. Our implementation of subdivision was on the software side of the pipeline, being executed by the CPU. This caused a performance hit that could be avoided by moving the calculations to either the tessellation stage in the forward pipeline, or possibly a compute shader, allowing the same
calculation to take place but utilising the GPU’s power to perform the calculations.

5.2 Known Issues and Improvements

There are a number of issues that, as the project developed, became clear and grew into problems that would ideally be fixed. The first we wish to address is the subdivision. As previously mentioned, running the subdivision on the CPU negatively impacts performance, and could be greatly improved upon by moving it to either the compute shader or the tessellation stage of the pipeline. Our subdivision implementation also doesn’t take into account whether a point already exists, resulting in redundant data being copied to the vertex buffer.

Our implementation of Back Face Culling is the heaviest function in the engine in terms of performance. This is caused by the loop for each potential vertex to check if it already exists in the buffer already. This system could be implemented into the subdivision, but the performance impact wasn’t worth eliminating the redundancy.

Our implementation also doesn’t account for holes in the surface of the object. Instead, the subdivision is just repeated until the hole is filled, causing areas that are already complete to gain points that are not necessary. If time permitted, a hole detecting algorithm would be devised and, since splitting objects is already a trivial task, splitting the area into a sub-object and subdividing it alone would help solve the problem.

The splatting techniques we devised and implemented have a number of issues inherent in their design which, if addressed, could greatly improve upon the visual quality of the renders produced. The chief among these issues is the placing of the surfels themselves. In our implementation, we plat in image space and do not orient the surfel. However, orienting the surfel based on the normal of the source point would greatly improve the overall visual quality, particularly for the rectangles which produce an aliasing effect at the edges of the model.

The other main issue with the splatting techniques is the size of the surfel produced. Currently, our implementation has a hard-coded size for the surfels that require it. How-
ever, dynamically determining the size would greatly help, but this would rely on a knowledge of neighbouring points, and intelligently choosing a size that would cover at least half the distance. The need for the surfel to cover at least half is derived from the possibility of blending taking place (which would mean an overlap is needed) and a possibility for distances between points to not be uniform, meaning surfels would be generated of different sizes. This could potentially introduce holes if the surfel does not cover at least half the distance to it’s nearest neighbour.

5.3 Final Thoughts

This project has taught me a lot, not only about the subject area, but graphics programming in general. The resulting boost in knowledge is not trivial, and actually supported my general programming development and C++ learning. It is my hope that this area will be investigated further, and my work expanded upon and improved by anyone with an interest and a desire.
6 Bibliography

References


Appendices

A

Machine Specification

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core i5-3570k</td>
</tr>
<tr>
<td>CPU Speed</td>
<td>3410 Mhz</td>
</tr>
<tr>
<td>RAM</td>
<td>16339 Mb</td>
</tr>
<tr>
<td>Graphics Card</td>
<td>NVIDIA GeForce GTX 670</td>
</tr>
<tr>
<td>Graphics VRAM</td>
<td>2047 Mb</td>
</tr>
<tr>
<td>Screen Resolution</td>
<td>1920x1080</td>
</tr>
</tbody>
</table>
Control Scheme

<table>
<thead>
<tr>
<th>Operation</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recompile Shaders</td>
<td>F1</td>
</tr>
<tr>
<td>Reset Camera</td>
<td>F2</td>
</tr>
<tr>
<td>Reload Scene</td>
<td>F3</td>
</tr>
<tr>
<td>Camera Fly Up</td>
<td>W</td>
</tr>
<tr>
<td>Camera Fly Down</td>
<td>S</td>
</tr>
<tr>
<td>Strafe Left</td>
<td>A</td>
</tr>
<tr>
<td>Strafe Right</td>
<td>D</td>
</tr>
<tr>
<td>Zoom In/Out</td>
<td>Mouse Wheel</td>
</tr>
<tr>
<td>Subdivide</td>
<td>Up Arrow</td>
</tr>
<tr>
<td>Decimate</td>
<td>Down Arrow</td>
</tr>
<tr>
<td>Increase Rotation Speed</td>
<td>+</td>
</tr>
<tr>
<td>Decrease Rotation Speed</td>
<td>-</td>
</tr>
<tr>
<td>Rotate on X Axis</td>
<td>1</td>
</tr>
<tr>
<td>Rotate on Y Axis</td>
<td>2</td>
</tr>
<tr>
<td>Rotate on Z Axis</td>
<td>3</td>
</tr>
<tr>
<td>Manually Rotate on Y Axis</td>
<td>Middle Mouse(Click and Drag)</td>
</tr>
<tr>
<td>Toggle Render Mode</td>
<td>Tab</td>
</tr>
<tr>
<td>Quit</td>
<td>ESC</td>
</tr>
</tbody>
</table>

Geometry Shader

```c
static const float PI = 3.14159265f;

struct GSInput
{
    float4 pos : SV_POSITION;
    float4 worldPosition : TEXCOORD3;
    float4 color : COLOR;
    float4 shadow : SHADOW;
    float4 positionsVsLight : TEXCOORD2;
    float3 normal : NORMAL;
    float2 texCoord : TEXCOORD;
};
```
float3 tangent : TANGENT;
float3 binormal : BINORMAL;
};

struct GSOutput
{
    float4 pos : SV_POSITION;
    float4 worldPosition : TEXCOORD3;
    float4 color : COLOR;
    float4 shadow : SHADOW;
    float4 positionsVsLight : TEXCOORD2;
    float3 normal : NORMAL;
    float2 texCoord : TEXCOORD;
    float3 tangent : TANGENT;
    float3 binormal : BINORMAL;
};

// // POINT //
// [maxvertexcount(1)]
void GShader(point GSInput input[1] : SV_POSITION, inout PointStream<GSOutput> output)
{
    GSOutput element;
    element.pos = input[0].pos;
    element.worldPosition = input[0].worldPosition;
    element.color = input[0].color;
    element.shadow = input[0].shadow;
    element.positionsVsLight = input[0].positionsVsLight;
    element.normal = input[0].normal;
    element.texCoord = input[0].texCoord;
    element.tangent = input[0].tangent;
    element.binormal = input[0].binormal;
    output.Append(element);
}

// // DISK //
// [maxvertexcount(32)]
void GShader(point GSInput input[1] : SV_POSITION, inout TriangleStream<GSOutput> output)
{
    GSOutput element;
    float x;
    float y;
    float radius = 0.021f;
    element.pos = input[0].pos;
element.worldPosition = input[0].worldPosition;
element.color = input[0].color;
element.shadow = input[0].shadow;
element.positionsVsLight = input[0].positionsVsLight;
element.normal = input[0].normal;
element.texCoord = input[0].texCoord;
element.tangent = input[0].tangent;
element.binormal = input[0].binormal;
output.Append(element);

float step = (2 * PI) / 16;

for (float angle = 0.0f; angle <= 2 * PI; angle += step) {
    element.pos.x = input[0].pos.x + radius * cos(angle);
    element.pos.y = input[0].pos.y + radius * sin(angle);
    element.texCoord.x = (input[0].texCoord.x + radius * cos(angle) * radius);
    element.texCoord.y = (input[0].texCoord.y + radius * sin(angle) * radius);
    output.Append(element);

    element.pos = input[0].pos;
    element.texCoord.x = input[0].texCoord.x;
    element.texCoord.y = input[0].texCoord.y;
    output.Append(element);
}

// Output first generated position to complete disk
element.pos.x = input[0].pos.x + radius * cos(0.0f);
element.pos.y = input[0].pos.y + radius * sin(0.0f);
element.texCoord.x = (input[0].texCoord.x + radius * cos(0.0f) * radius);
element.texCoord.y = (input[0].texCoord.y + radius * sin(0.0f) * radius);
output.Append(element);
output.RestartStrip();

//
// Rectangle
//
[maxvertexcount(5)]
void GShader(point GSInput input[1] : SV_POSITION, inout 
TriangleStream<GSOutput> output) {
    GSOutput element;

    float distance = 0.05f;
float texDis = distance / 2048;
element.pos = input[0].pos;
element.worldPosition = input[0].worldPosition;
element.color = input[0].color;
element.shadow = input[0].shadow;
element.positionsVsLight = input[0].positionsVsLight;
element.normal = input[0].normal;
element.texCoord = input[0].texCoord;
element.tangent = input[0].tangent;
element.binormal = input[0].binormal;
output.Append(element);

//for now just do it on x, use set distance (square)
//TODO: Orientation will determine axis
//TODO: Size should be calculated based on surrounding area and needs

//top left
element.pos.x = input[0].pos.x - distance;
element.pos.y = input[0].pos.y + distance;
element.texCoord.x = input[0].texCoord.x - texDis;
element.texCoord.y = input[0].texCoord.y + texDis;
output.Append(element);

//top right
element.pos.x = input[0].pos.x + distance;
element.pos.y = input[0].pos.y + distance;
element.texCoord.x = input[0].texCoord.x + texDis;
element.texCoord.y = input[0].texCoord.y + texDis;
output.Append(element);

//bottom left
element.pos.x = input[0].pos.x - distance;
element.pos.y = input[0].pos.y - distance;
element.texCoord.x = input[0].texCoord.x - texDis;
element.texCoord.y = input[0].texCoord.y - texDis;
output.Append(element);

//bottom right
element.pos.x = input[0].pos.x + distance;
element.pos.y = input[0].pos.y - distance;
element.texCoord.x = input[0].texCoord.x + texDis;
element.texCoord.y = input[0].texCoord.y - texDis;
output.Append(element);
void Renderable::BackFaceCull(XMFLOAT3 camPos)
{
    originalIndices = indices;
    originalVertices = vertices;
    vertices.clear();
    indices.clear();

    XMFLOAT3 polyNormal;
    XMFLOAT3 worldCam;
    XMFLOAT3 viewVec;
    VERTEX one;
    VERTEX two;
    VERTEX thr;

    XMFLOAT3 vectorA;
    XMFLOAT3 vectorB;

    float dot;

    //Loop through the triangles
    for (unsigned int i = 0; i < originalIndices.size(); i += 3)
    {
        //Assign the three vertices of the triangle
        one = originalVertices[originalIndices[i]];
        two = originalVertices[originalIndices[i + 1]];
        thr = originalVertices[originalIndices[i + 2]];

        //Calculate the polygon’s normal
        vectorA = Maths.MinusVectors(two.Position, one.Position);
        vectorB = Maths.MinusVectors(thr.Position, one.Position);
        polyNormal = Maths.CrossProduct(vectorA, vectorB);

        viewVec = camPos;
        Maths.Normalise(polyNormal);
        Maths.Normalise(viewVec);
        dot = Maths.DotProduct(polyNormal, viewVec);
        if (dot >= 0.0f)
        {
            //facing camera

            //Horribly inefficient
            int posOne = std::find(vertices.begin(), vertices.end(),
                                   one) - vertices.begin();
            if (!(posOne < vertices.size()))
            {
                vertices.push_back(one);
            }
        }
    }
}
posOne = vertices.size() - 1;
}
int posTwo = std::find(vertices.begin(), vertices.end(),
    two) - vertices.begin();
if (!(posTwo < vertices.size()))
{
    vertices.push_back(two);
    posTwo = vertices.size() - 1;
}
int posThree = std::find(vertices.begin(),
    vertices.end(), thr) - vertices.begin();
if (!(posThree < vertices.size()))
{
    vertices.push_back(thr);
    posThree = vertices.size() - 1;
}
indices.push_back(posOne);
indices.push_back(posTwo);
indices.push_back(posThree);
}
Subdivision

```cpp
void Renderable::SubDivide(int iterations)
{
    VERTEX one;
    VERTEX two;
    VERTEX thr;

    VERTEX left;
    VERTEX right;
    VERTEX bottom;

    int vertInd6 = 0;
    int vertInd5 = 0;
    int vertInd4 = 0;
    int vertInd3 = 0;
    int vertInd2 = 0;
    int vertInd1 = 0;

    bool present6 = false;
    bool present5 = false;
    bool present4 = false;
    bool present3 = false;
    bool present2 = false;
    bool present1 = false;

    std::vector<VERTEX> tempVerts;
    std::vector<int> tempIndices;

    int originalMax = indices.size();
    for (int index = 0; index < originalMax; index += 3)
    {
        vertInd6 = 0;
        vertInd5 = 0;
        vertInd4 = 0;
        vertInd3 = 0;
        vertInd2 = 0;
        vertInd1 = 0;
        present6 = false;
        present5 = false;
        present4 = false;
        present3 = false;
        present2 = false;
        present1 = false;

        one = vertices[indices[index]];
        two = vertices[indices[index + 1]];
    }
}
```
thr = vertices[indices[index + 2]];
/*
  1
 /\       
/  \
/    \_____
 2     3
BECOMES
  \/
 /__
/  /__
/__/__/__/ 
TRIFORCE
*/

left = one;
right = one;
bottom = one;

//get left
left.Position.x = (one.Position.x + two.Position.x) / 2;
left.Position.y = (one.Position.y + two.Position.y) / 2;
left.Position.z = (one.Position.z + two.Position.z) / 2;
left.TexCoord.x = (one.TexCoord.x + two.TexCoord.x) / 2;
left.TexCoord.y = (one.TexCoord.y + two.TexCoord.y) / 2;

//get right
right.Position.x = (one.Position.x + thr.Position.x) / 2;
right.Position.y = (one.Position.y + thr.Position.y) / 2;
right.Position.z = (one.Position.z + thr.Position.z) / 2;
right.TexCoord.x = (one.TexCoord.x + thr.TexCoord.x) / 2;
right.TexCoord.y = (one.TexCoord.y + thr.TexCoord.y) / 2;

//get bottom
bottom.Position.x = (two.Position.x + thr.Position.x) / 2;
bottom.Position.y = (two.Position.y + thr.Position.y) / 2;
bottom.Position.z = (two.Position.z + thr.Position.z) / 2;
bottom.TexCoord.x = (two.TexCoord.x + thr.TexCoord.x) / 2;
bottom.TexCoord.y = (two.TexCoord.y + thr.TexCoord.y) / 2;

tempVerts.push_back(one);       // - 6
tempVerts.push_back(two);       // - 5
tempVerts.push_back(thr);       // - 4
tempVerts.push_back(left);      // - 3
tempVerts.push_back(right);     // - 2
tempVerts.push_back(bottom);    // - 1

vertInd6 = tempVerts.size() - 6;
vertInd5 = tempVerts.size() - 5;
vertInd4 = tempVerts.size() - 4;
vertInd3 = tempVerts.size() - 3;
vertInd2 = tempVerts.size() - 2;
vertInd1 = tempVerts.size() - 1;

tempIndices.push_back(vertInd2);    // top
tempIndices.push_back(vertInd3);
tempIndices.push_back(vertInd6);

tempIndices.push_back(vertInd4);    // bottom left
tempIndices.push_back(vertInd1);
tempIndices.push_back(vertInd2);

tempIndices.push_back(vertInd1);    // center
tempIndices.push_back(vertInd3);
tempIndices.push_back(vertInd2);

tempIndices.push_back(vertInd1);    // bottom left
tempIndices.push_back(vertInd5);
tempIndices.push_back(vertInd3);

}
vertices.clear();
indices.clear();
vertices = tempVerts;
indices = tempIndices;
}
F

Decimation

```c++
void Renderable::Decimate(int iterations)
{
    VERTEX one;
    VERTEX two;
    VERTEX thr;

    std::vector<VERTEX> tempVerts;
    std::vector<int> tempIndices;

    int originalMax = indices.size();
    for (int index = 0; index < originalMax; index += 12)
    {
        one = vertices[indices[index + 2]];
        two = vertices[indices[index + 10]];
        thr = vertices[indices[index + 3]];

        tempVerts.push_back(one);
        tempVerts.push_back(two);
        tempVerts.push_back(thr);

        tempIndices.push_back(tempVerts.size() - 3);
        tempIndices.push_back(tempVerts.size() - 2);
        tempIndices.push_back(tempVerts.size() - 1);
    }
    vertices.clear();
    indices.clear();
    vertices = tempVerts;
    indices = tempIndices;
}
```
Object Splitting

```cpp
template <class T>
void SplitVectors(ObjLoader* loader, int splits, std::vector<Renderable*>& objects, float scale, LPWSTR mat, LPWSTR bump, LPWSTR height)
{
    std::vector<VERTEX> tempVerts;
    std::vector<int> tempIndices;

    int one = 0;
    int two = 0;
    int thr = 0;

    unsigned int threshold = loader->indices.size() / splits;
    int step = threshold;

    for (unsigned int i = 0; i < loader->indices.size(); i += 3)
    {
        if (i >= threshold)
        {
            T* obj;
            if (height != L""")
            {
                obj = new T(mat, bump, height, tempVerts, tempIndices, scale);
            }
            else
            {
                obj = new T(mat, bump, tempVerts, tempIndices, scale);
            }
            AddToScene(obj, bufferDesc, ms, 0.f, 0.f, 0.f);
            tempVerts.clear();
            tempIndices.clear();
            threshold += step;
        }

        one = loader->indices[i];
        two = loader->indices[i + 1];
        thr = loader->indices[i + 2];

        //vertices
        tempVerts.push_back(loader->vertices[one]);
        tempVerts.push_back(loader->vertices[two]);
        tempVerts.push_back(loader->vertices[thr]);

        //indices
        tempIndices.push_back(tempVerts.size() - 3);
    }
}
```
tempIndices.push_back(tempVerts.size() - 2);
  tempIndices.push_back(tempVerts.size() - 1);
}

// Put any spill-over into an object too.
if (tempVerts.size() > 0)
{
  T* obj;
  if (height != L""")
  {
    obj = new T(mat, bump, height, tempVerts, tempIndices,
                 scale);
  }
  else
  {
    obj = new T(mat, bump, tempVerts, tempIndices, scale);
  }
  AddToScene(obj, bufferDesc, ms, 0.f, 0.f, 0.f);
}

tempVerts.clear();
tempIndices.clear();
Figure 12: Our Engine Class Diagram
Spider-Man Subdivision Iterations Using our Three Splatting Techniques

Figure 13: Textured Spider-Man Using Points
Figure 14: Textured Spider-Man Using Disks

(a) First Iteration  
(b) Second Iteration  
(c) Third Iteration  
(d) Fourth Iteration

Figure 15: Textured Spider-Man Using Rectangles

(a) First Iteration  
(b) Second Iteration  
(c) Third Iteration  
(d) Fourth Iteration