DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Real-time rendering of large point clouds

Alexandru-Ștefan Todoran

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Real-time rendering of large point clouds

Echtzeit Rendering von hochaufgelösten Punktwolken

Author:	Alexandru-Ștefan Todoran
Supervisor:	Prof. Dr. Rüdiger Westermann
Advisor:	Prof. Dr. Rüdiger Westermann
Submission Date:	15.03.2019

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.03.2019 Alexandru-Ștefan Todoran

Contents

Abstract

Point clouds compression is an interesting and important topic, since efficiently compressing point clouds, enables the visualisation of larger datasets. This thesis examines two methods of compression and performs an analysis on their performance. To this end, a renderer implementation is presented, on which benchmarks of the methods were run. Using the obtained benchmarks we show that the traditional method is superior.

1 Introduction

Point clouds are a widely used representation for 3d scene data. They are usually obtained from scanned surface data, and are being used in a variety of fields, from geographical information systems to urban planning. One problem, however, is that the datasets obtained from a laser scan can contain a large number of points, and thus, be very memory intensive.

In this thesis, we are interested in compressing point clouds that represent surfaces of objects, in contrast to point clouds that represent volumetric data, such as from medical scans, or particle simulations. We look at two compressed representations of such point clouds and compare their memory consumption and decoding performance against a baseline implementation.

We will begin by looking at what is needed to render a point cloud, and how such a dataset is stored in general. We will be examining the size of a typical point cloud (per point), and will be looking at ways of reducing the memory footprint. Afterwards, the design of an implementation program written in C++is described, along with a short user guide on the program itself. Then, the different approaches are described in practice, and their viability evaluated. Auxiliary techniques such as Level of detail (LOD) and occlusion culling, that were not inspected in close enough detail but might further improve performance are described in the future work section.

2 Point cloud rendering

2.1 Lighting

A point cloud is an unstructured collection of points, or vertices. At a minimum, each vertex contains information about its position in 3d space. With only positions at our disposition, the most we can do, is draw pixels of an arbitrary color at each position. Figure 2.1 shows a point cloud scan of castle Neuschwanstein, made up of aproximatively one million points. The dataset itself also contains normals and colors, but we only take positions into account at first. Even this render already gives a rough sense of scale and we get an impression of the castle's silhouette. It is hard, however to infer how the different features are placed in relation to each other, just from a still image. There is no lighting to tell us (through shadows) whether the tower on the left, for example, lies behind or in front of the castle.

When normals are available, we can think of each vertex in a point cloud as representing an oriented disk with a finite area. If we define a light source through a position and a light color, we now have everything we need to shade the resulting disk.

We will shade each disk with the so called "Blinn-Phong" lighting model([1]). Figure 2.2 shows the unit vectors relevant for the lighting calculation, as seen from a single point on a disk. $\vec{N}$ is the normal of the current disk, $\vec{L}$ is the vector pointing towards the light source and $\vec{V}$ points towards the camera. The reflection vector $\vec{R}$ shows the direction in which the light would be reflected if the surface was a perfect mirror. The halfway vector is defined as $\vec{H} = \frac{\vec{V}+\vec{L}}{|\vec{V}+\vec{L}|}$. In the Blinn-Phong model, the lighting contribution is split into three parts: the diffuse, the specular, and the ambient lighting contribution.

The diffuse lighting contribution represents light that is scattered uniformly around the normal, without taking the view direction $\vec{V}$ into account. It is calculated as follows: $lighting_{diffuse} = color_{material} \cdot max(\vec{N} \cdot \vec{L}, 0)$. The more aligned with the light direction the normal is, the more intense the surface is lit up. If the dot product is negative (i.e. the normal faces away from the light), the diffuse component is 0, and thus, completely black. Figure 2.3

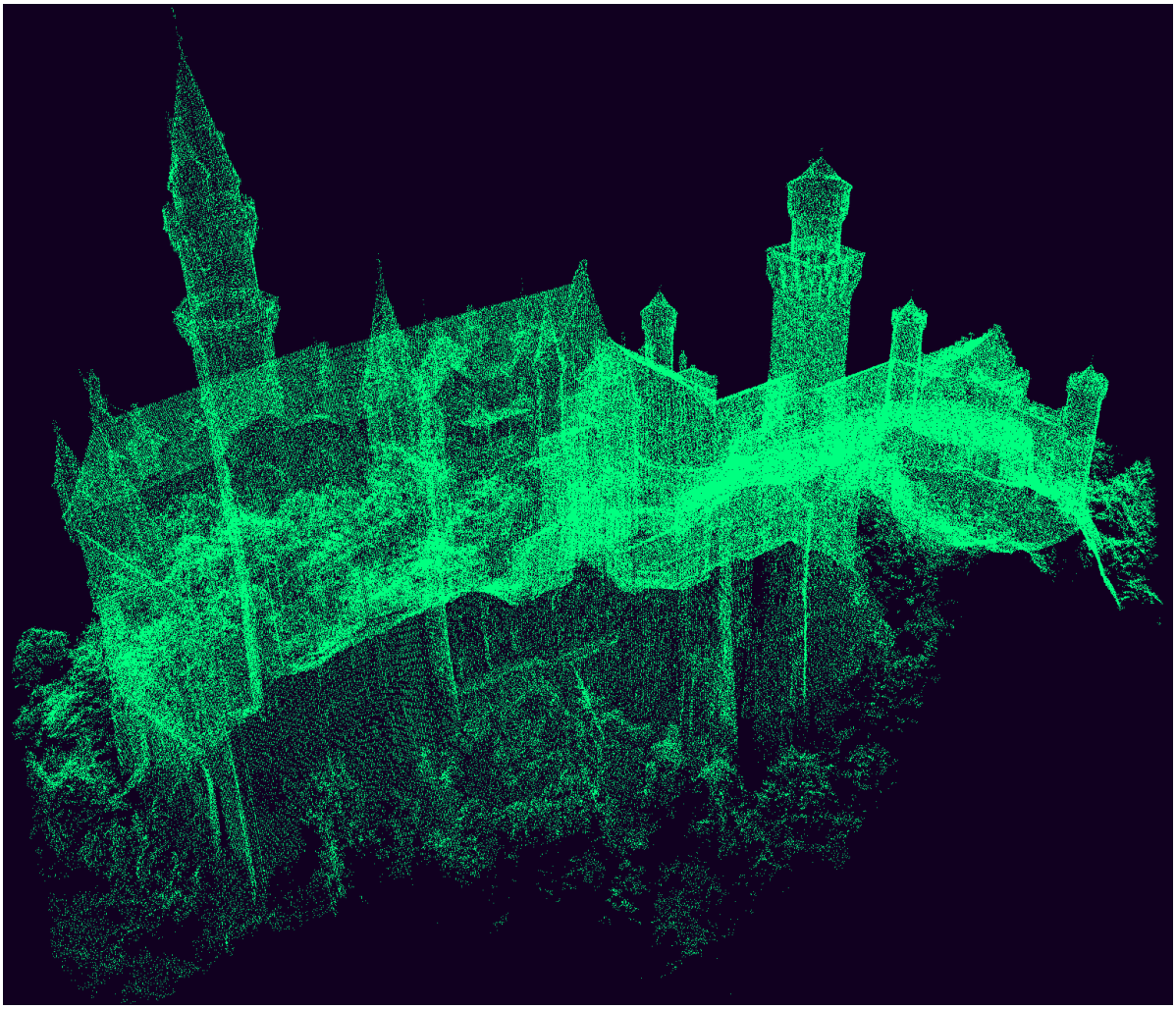

Figure 2.1: Positions only

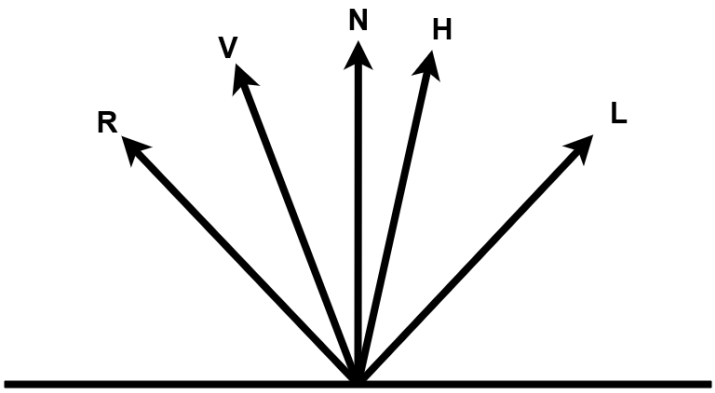

Figure 2.2: Vectors used for shading in the blinn-phong lighting model

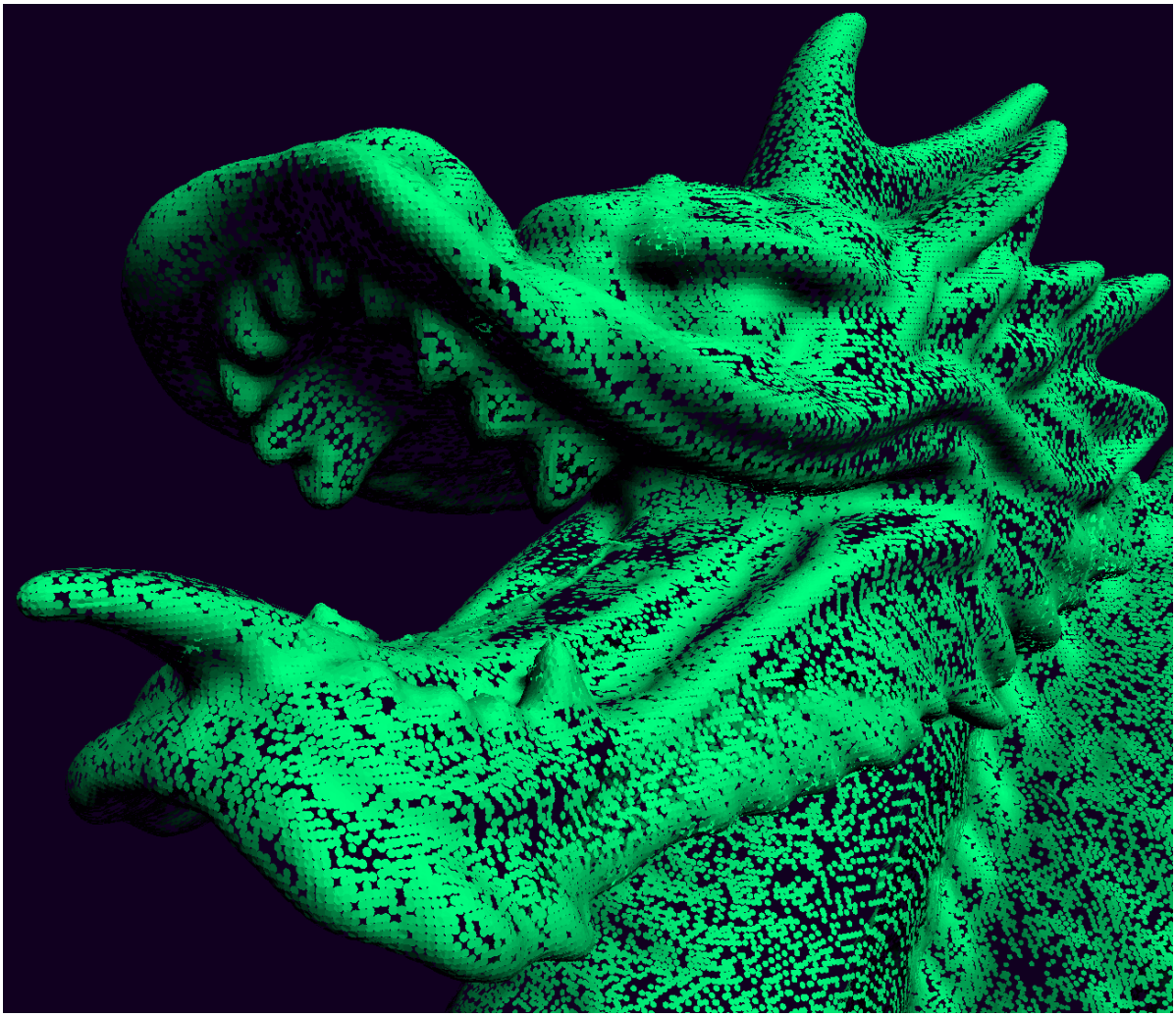

Figure 2.3: Diffuse lighting contribution

shows the stanford dragon with diffuse lighting.

Figure 2.4: Specular lighting contribution with increasing shininess factor

The specular lighting contribution represents the light that gets reflected around the reflection vector $\vec{R}$. Intuitively, this component simulates the highlights on shiny materials. The following formula is used to calculate it: $lighting_{specular} = color_{specular} \cdot (max(\vec{N} \cdot \vec{H}, 0))^{factor_{shininess}}$. In this formula, we are interested in the dot product between the normal $\vec{N}$ and the halfway vector $\vec{H}$. The more these two vectors align, the more the view direction $\vec{V}$ aligns with the direction of the reflected light $\vec{R}$, and thus, more light hits the eye. The dot product is raised to a shininess factor. This factor allows us to influence the "spread" of the specular highlight, and, thus control how "shiny" the material looks. The $color_{specular}$ represents the color of the highlight itself. This is usually white, although metallic materials can have colored highlights (e.g. gold or copper). Figure 2.4 shows the specular contribution with the shininess factor varied (from left to right:8, 16, and 64).

The contributions are used in the following formula to obtain the final color of the surface: $color_{final} = color_{light} \cdot (lighting_{diffuse} + lighting_{specular}) + color_{ambient}$. The diffuse and specular contributions are added together and multiplied by the color of the light itself, since it also contributes to the final color of the surface. A fixed ambient color is added, regardless of surface orientation or lighting, to simulate indirect lighting. In figure 2.5 in the second panel, the final lighting can be seen, with both diffuse and specular lighting taken into account.

As can be seen in figure 2.6, normals allow us to give a better sense of depth and surface orientation to the point cloud. From a distance, we get the illusion that we are looking at continuous surfaces. Figure 2.7 shows colors applied in addition to shading. The impact of colors seems to be smaller, and mostly aesthetic, compared to the impact of having normals present in a dataset.

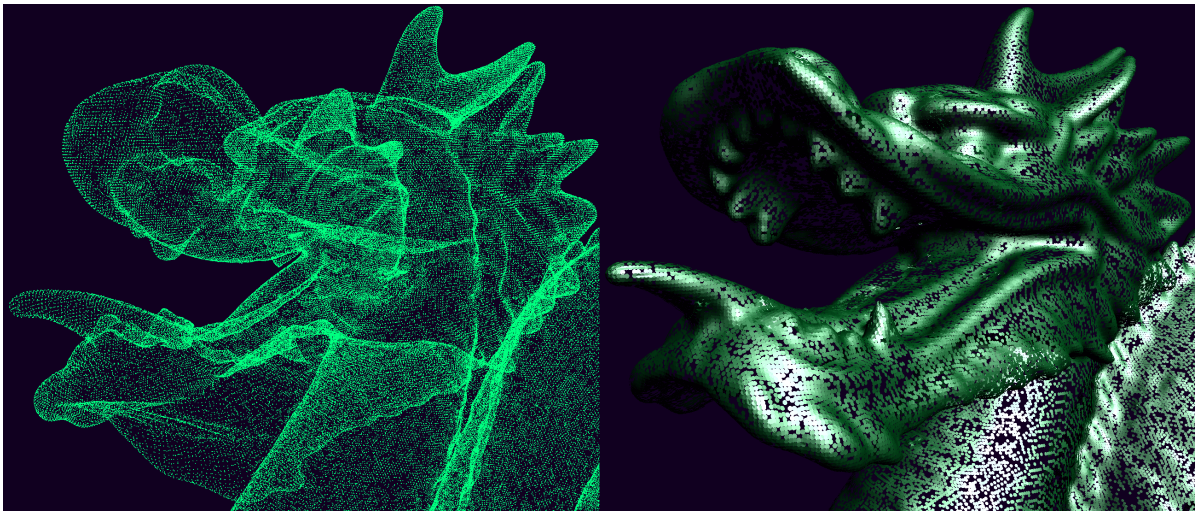

Figure 2.5: Positions only(left) vs fully shaded(right)

2.2 Representation

2.2.1 Positions

Each vertex typically stores its position as 3 32bit floating point numbers, one for each axis. This results in 96 bits per vertex just for storing the positions.

Bricks

One possibility for reducing the needed amount of memory per vertex, is subdividing the bounding box of the point cloud into equally sized "bricks". Each brick contains a subset of the point clouds' vertices and has a length, width and depth. Each vertex would then only need to store its relative position inside the parent brick. Less precision is needed for the relative positions, since they cover a smaller space than the global positions. Regular unsigned 8 bit integers could be used, which would result in a resolution of 256^3 possible locations per brick.

In order to get back the absolute position of a vertex, the following calculation needs to be performed: $p_{absolute} = p_{brick} + p_{relative} \cdot size_{brick}$. The brick position could be stored in full precision, or it could be recalculated from the index of the brick: $size_{brick} = index_{brick} \cdot size_{brick}$. This would reduce the memory consumption per brick as well, as the brick index can then be

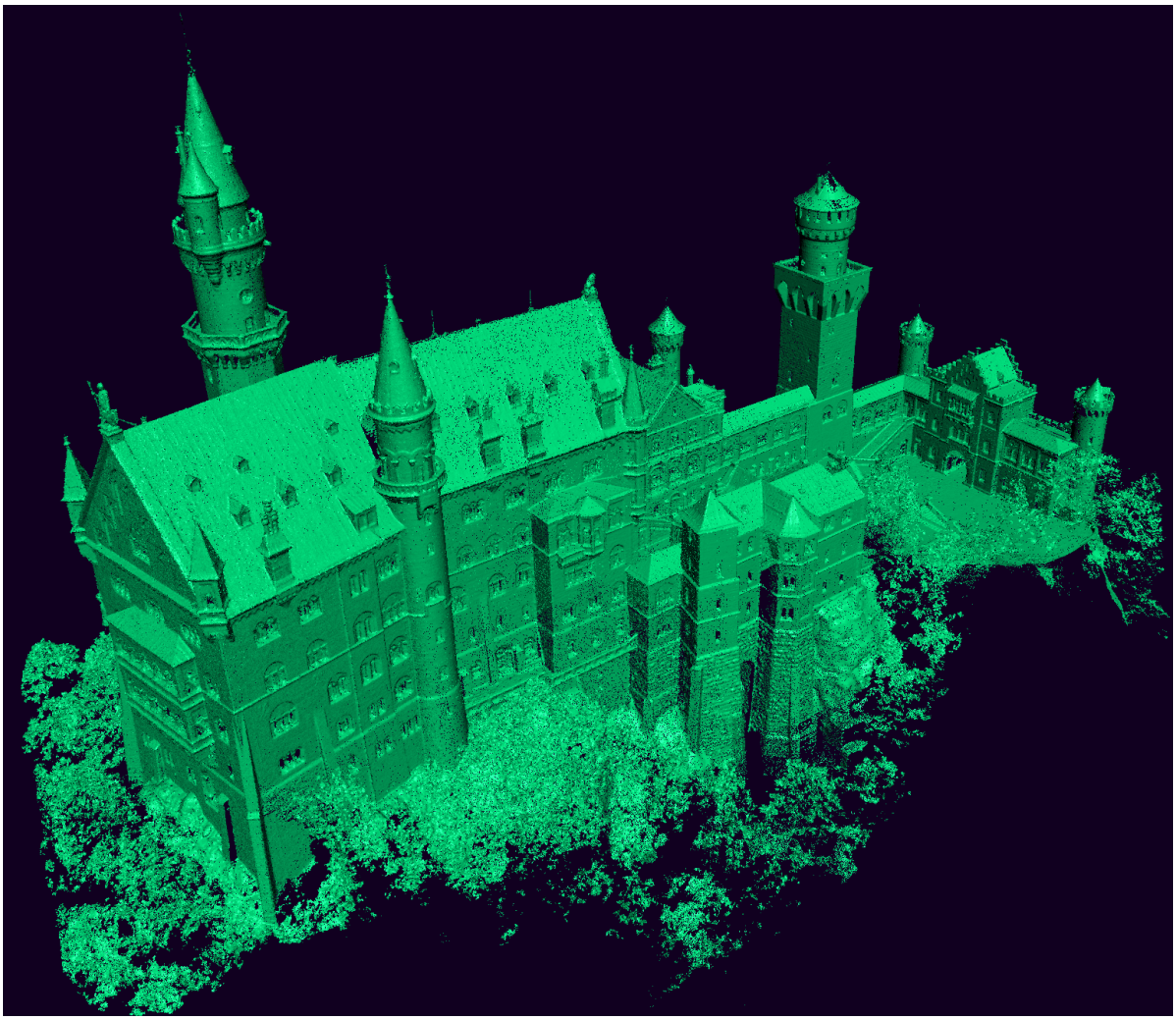

Figure 2.6: Positions and normals

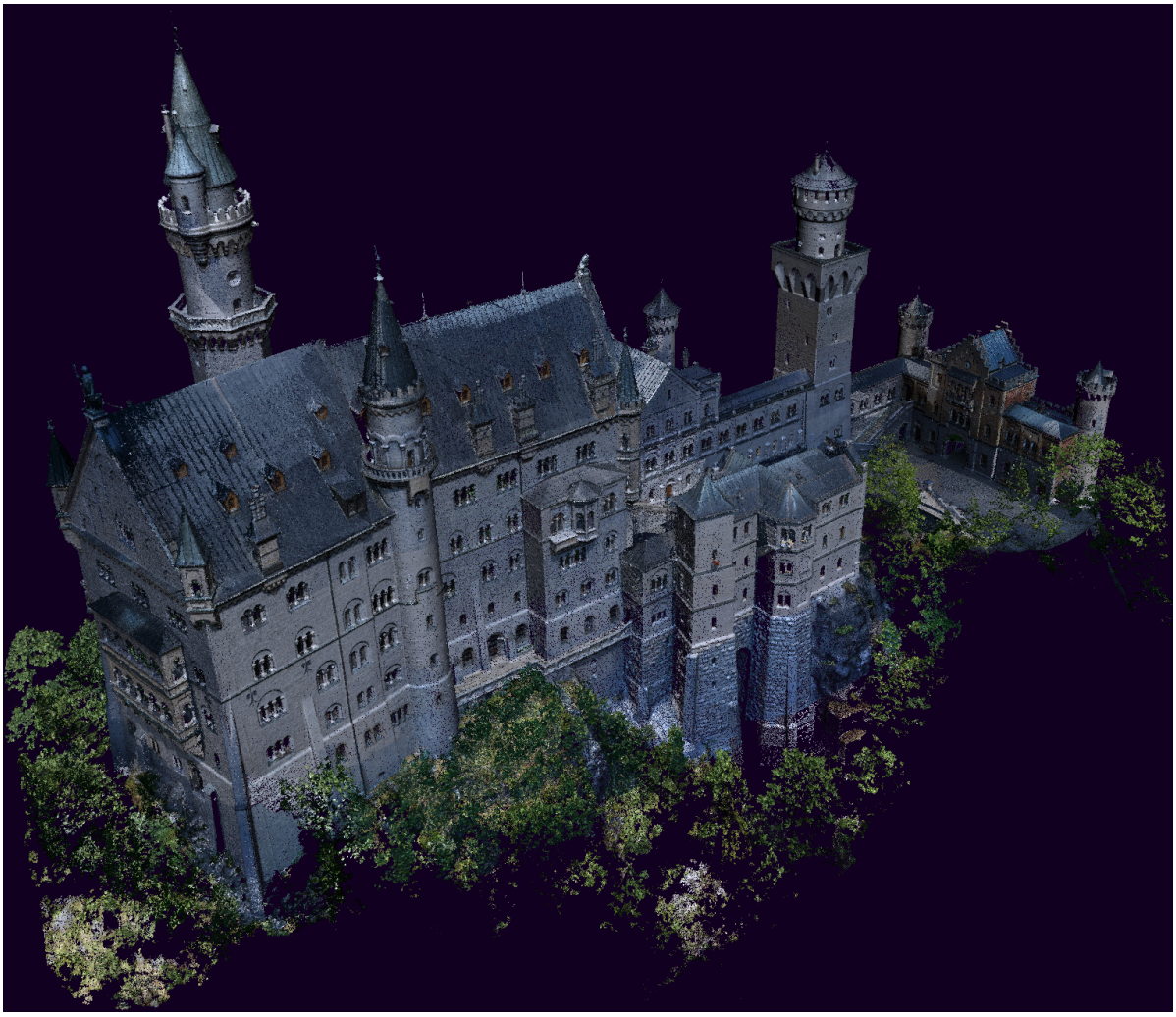

Figure 2.7: Position, normals and color data

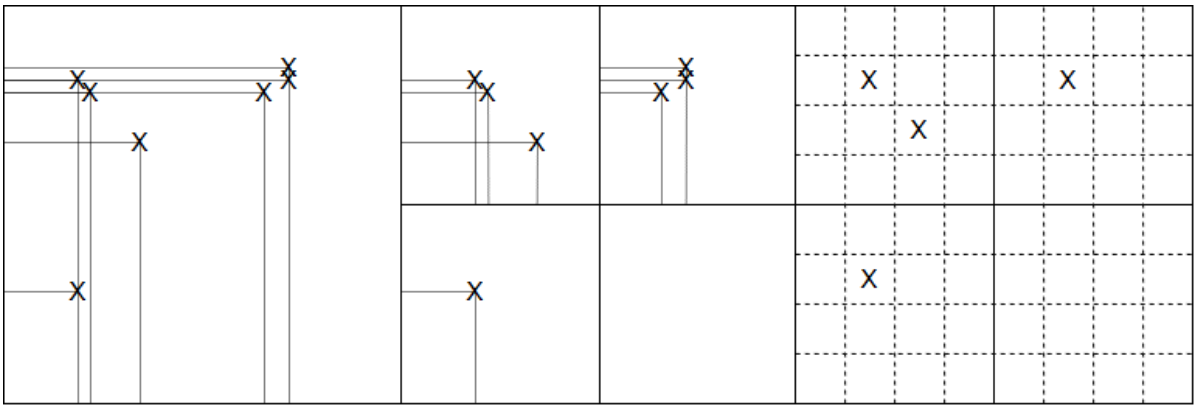

Figure 2.8: Left to right: uncompressed, bricks, bitmaps

stored in a single integer. For example, we could index up to approximately $512^3 = 2^{30}$ bricks using a 32bit unsigned integer per brick.

Figure 2.8 shows the different forms of storing positions. On the left is the uncompressed form. In the middle is the brick subdivision method discussed previously. The thick lines represent the individual lines, whereas the thin lines represent the offsets that are stored per vertex. Since the offsets lines for the brick subdivision method are shorter, less precision is needed per offset.

Bitmaps

The third method, visible in the third panel of figure 2.8 is the bitmap method. The point cloud is still subdivided into bricks of equal size, however, instead of storing numbers that represent the offsets of individual vertices, a bitmap is stored, where each bit represents one possible position inside the brick. A set bit tells us that that position is occupied, and a vertex exists there. This results in a fixed memory usage per brick, independent of vertex count. When decompressing the bitmap and encountering a set bit, let i be the index of the current bit within the bitmap, and s be the size of one side of the bitmap(the bitmap has s^3 bits). Then, the following formulas can be used to decode the offset within the brick: $p = ((i \bmod s^2) \bmod s, \frac{i \bmod s^2}{s}, \frac{i}{s^2})$

In the figure, the dashed lines delimit the possible positions, or the bits of the bitmap. For the upper-left brick, by iterating over the possible positions left to right, top to bottom, the following bitmap would be generated: `0000 0100 0010 0000`.

Two problems are apparent when using this method. Firstly, memory is used even for the empty positions. This means that there needs to be a high number of vertices present in the brick, so that the fixed memory occupied by the bitmap is more efficient than just storing the offsets per vertex. Secondly, if the bitmap resolution is not high enough, multiple points will be represented by a single position. This can be seen in the figure, where 3 points are lost in total this way. In order to prevent data loss, either the bitmap resolution or the brick subdivision count would need to be increased. The brick offset method suffers from this problem as well, however, the impact in memory does not increase as fast as in the bitmap case, when greater precision is used. This fact is discussed in more detail in chapter 4.

2.2.2 Normals

Similarly to positions, normals are stored as 3 component vectors of floating point numbers. However, a normal and a vector are not exactly the same. A normal always has unit length, meaning that we can think of normals as points on a sphere. However, floating point numbers encode the full $\mathbb{R}^3$ space. Cigolle et al. discuss this problem in more detail and examine a multitude of encoding schemes for normals specifically in [2].

For our purposes, however, spherical coordinates represent a straightforward way to reduce the memory footprint of normals. The vector representation is converted to two angles $\phi \in [-\pi, \pi]$ and $\theta \in [0, \pi]$ first, using the following formulas: $\phi = \arctan(n.x, n.z), \theta = \arccos(n.y)$. If we imagine the normal as a vector with the origin at the center of a sphere, then ϕ is the angle of the normal alongside the equator (also called azimuth), and θ (zenith) is the angle perpendicular to ϕ.

By further normalizing these two angles, we obtain a 2-dimensional vector, s whose components each span the range $[0, 1] : s = (\frac{\phi}{2\pi} + 0.5, \frac{\theta}{\pi})$. For this vector, it is then trivial to use any precision desired. In the implementation of this thesis, 8 bits per component, for a total of 16 bits per normal have been used, without any noticeable degradation in quality.

3 Implementation

The rendering engine implemented for this thesis is called "LPCRenderer" (for Large point cloud renderer). The engine was developed using C++17 and Microsoft Visual Studio 2017. Binaries are provided for x64 Windows. No build scripts are provided for other platforms, however the code itself does not rely on any Windows specific behaviour, and uses cross-platform libraries for things such as window handling and OpenGL function pointer loading.

Section 3.1 will describe how the software is structured as well as what third party libraries are used. Afterwards, section 3.2 will explain the role of each window and show a typical workflow. In section 3.3, the different implementations of point cloud rendering and compression are presented and discussed.

3.1 Engine architecture

Some components of the engine (e.g. 3.1.1, 3.1.2) are implemented as free functions in a namespace. The public interface is exposed in a header file, and the implementation details are all moved in a source file with the same name. From an object oriented point of view, these components would be implemented as singletons, since there is no point in having more than one instance of such a component. For resource management, providing a template for constructing managers in charge of a particular type of resource made sense. In that case, the components still behave as singletons, but static template instances are used, instead of namespaces. Subsection 3.1.3 talks about this in more detail.

3.1.1 Window handling

A rendering engine will typically take some kind of input (scene description, lighting parameters, resolution etc.) and output a framebuffer. The input could come from command line arguments, or from a configuration file, and the output framebuffer could be written to

```
1  OSWindow::init();
2  while(!OSWindow::shouldClose())
3  {
4          OSWindow::beginFrame();
5
6          Profiler::recordFrame();
7          MainRenderer::render(SceneManager::getActive());
8          drawUI();
9
10         OSWindow::endFrame();
11 }
12 OSWindow::destroy();
```

Figure 3.1: The main game loop

disk as an image each time the engine is invoked. Software that works in this way, without a graphical user interface, is usually called "headless". This is rather tedious, if one wants to experiment with different parameters and see the resulting changes to the image in real-time.

In order to move towards an interactive application, one first needs to draw the rendered framebuffer directly to a portion of the screen. For this, we need to talk to the operating system (OS). On OSes where a windowing system is present, an API is provided that allows us to create an OS window. Such a window has, at a minimum, a framebuffer where we can write our output to. It is common, however, for there to be 2 (or even more) framebuffers present. One framebuffer is considered to be the "frontbuffer" and the other one the "backbuffer". The front buffer is the one being displayed at that moment, and the back buffer is the one the application is currently drawing to. When we are finished with drawing the current frame, the front- and backbuffers are swapped, in order to display the updated image. This is done several times per second, depending on the capabilities of the computer and the complexity of the rendered image. Changes to rendering parameters or to the scene (camera position, lighting color, rendering modes etc.) are thus perceived instantly by the user and the illusion of continuity is obtained.

The component `OSWindow` is used to wrap such a window in a simple to use interface. The component itself does not use the underlying OS API directly, instead, `GLFW`, an open source and cross platform windowing library is used. Besides offering us a framebuffer we can draw to, `GLFW` also provides us with the ability to set callback functions that get called when certain events happen (e.g. window resizing or mouse and keyboard events).

Listing 3.1 shows the game loop, and the entirety of the main function for our program. In `OSWindow::init()`, a window is created, the respective callbacks are hooked up, and necessary libraries are initialized. The rendering loop itself is an infinite loop that checks each iteration whether `OSWindow::shouldClose()` returns true. This function returns true when the escape key has been pressed, or if a window close event has been fired by the operating system (e.g. when clicking the red 'X' in Windows 10). In `OSWindow::beginFrame()`, events are checked and handled, and the backbuffer is cleared, so that information from the previous frame is not left over. `OSWindow::endFrame()` finally swaps the buffers, thus showing the updated rendering. On program exit, `OSWindow::destroy()` tells the operating system that we are done with the window, and that it can be safely freed.

3.1.2 Model importing

The `ply` file format was created at Stanford and is capable of storing polygonal meshes with user defined per-vertex properties [3]. For our application we only support reading in vertices without connectivity information (so no triangle meshes, only point clouds). Each vertex should have positional data and, optionally, normal or color data. For parsing the `ply` file format, the C++11 header-only library, tinyply is used.

The `Importer` component can import a list of `ply` or `conf` files at once and merge them into a single point cloud mesh. All that `conf` files do, is reference other `ply` files, and offer a translation and rotation to apply to each contained point cloud. The raw scan data from the Stanford 3D Scanning Repository needs to be read in this way [4].

For representing vectors and matrices and doing math with them, the library glm is used. The most used class is `glm::vec3` and represents a vector of three floating point components. Other vector classes used throughout the program are:

- dvec*n* - 64 bit floating point
- ivec*n* - 32 bit signed integer
- u8vec*n* - 8 bit unsigned integer

The 3- and 4-component floating point matrix classes `glm::mat3` and `glm::mat4` are also used throughout the program. One major advantage of this library is that the author wrote it to be similar to the gl shading language (glsl), and it also mirrors some packing/unpacking functions that glsl supports, such as `packUnorm4x8` and `unpackUnorm4x8`. These functions

```
struct PointCloudBrick
{
        glm::ivec3 indices;
        std::vector<glm::vec3> positions;
        std::vector<glm::vec3> normals;
        std::vector<glm::u8vec3> colors;
};
class PointCloud
{
private:
        std::pair<glm::vec3, glm::vec3> bounds;
        glm::vec3 brickSize;
        std::vector<PointCloudBrick> bricks;
        std::size_t vertexCount = 0;
        glm::ivec3 subdivisions{0};
        ...
```

Figure 3.2: The intermediate representation of a point cloud

```
class PointCloud
{
private:
        ...
        mutable std::size_t emptyBrickCount = 0;
        mutable std::size_t redundantPointsIfCompressed = 0;
        mutable float pointsPerBrickAverage = 0;
        mutable std::size_t brickPrecision = 32;
public:
        void setBrickPrecision(std::size_t precision) const;
        void updateStatistics() const;
        glm::vec3 convertToWorldPosition(glm::ivec3 indices, glm::vec3 localPosition) const;
        std::pair<glm::vec3, glm::vec3> getBoundsAt(glm::ivec3 indices) const;
        glm::vec3 getOffsetAt(glm::ivec3 indices) const;
        std::unique_ptr<PointCloud> decimate(std::size_t maxPoints) const;
        void drawUI();
        ...
};
```

Figure 3.3: The helper methods provided by the point cloud class

are useful for compressing vectors of 4 32 bit floating point components into a single 32 bit unsigned integer, by reducing the precision of each component down to 8 bits and packing all four together [5].

The relevant parts of the internal point cloud representation are shown in 3.2. Each point cloud is stored as a vector of bricks. Only the non-empty bricks are stored, and each brick knows its index inside the 3D subdivision grid. The `normals` and `colors` members of `PointCloudBrick` are optional and can be empty.

Aside from the typical setters and getters, that are not shown, the point cloud class also offers helper methods for index math, and for calculating statistics related to points lost due to compression or average brick occupancy (3.3). The method on line 20 returns a copy of the current point cloud, with the point count kept under a specific limit. This is useful for testing even on weaker hardware.

3.1.3 Resource Management

```
1  template<typename T>
2  class Manager
3  {
4  private:
5          static inline std::vector<std::unique_ptr<T>> store{};
6  public:
7          static auto const& getAll();
8          static T* add(std::unique_ptr<T>&& resource);
9          T* getActive();
10         static void drawUI();
11         ...
12 };
```

Figure 3.4: Template for resource managers

For resource management, a helper template class exists. A resource manager is a wrapper around a static vector of pointers to resources of type *T*. Pointers are used, instead of storing the resource directly in the vector, so that references to a resource remain valid, even after adding new elements to the vector. The manager offers methods for adding a new resource instance, and for getting the currently active resource of that type, deleting resources, however, is not supported. Most importantly, this template offers a standard GUI for resource managers. This is explained further in 3.1.5.

One of the types considered a resource are point clouds themselves. After importing a point cloud, it is added to the `Manager<PointCloud>` class, and thus made visible to the rest of the program from that point on.

```
1  class Scene
2  {
3  private:
4          glm::vec3 backgroundColor{0.065f, 0.0f, 0.125f};
5          Camera camera;
6          glm::vec3 lightDirection = glm::normalize(glm::vec3{0.0f, -1.0f, -1.0f});
7          glm::vec3 lightColor{0.86f, 0.9f, 1.0f};
8          glm::vec3 diffuseColor{0.0f, 1.0f, 0.5f};
9          glm::vec3 specularColor{0.25f};
10         float shininess = 64.0f;
11         float ambientStrength = 0.05f;
12         PointCloud* cloud = nullptr;
13         glm::mat4 modelMatrix = glm::mat4(1.0f);
14         float scaling = 1.0f;
15 public:
16         void drawUI();
17         ...
18 };
```

Figure 3.5: The scene class

Another resource, and important class of the whole engine is the `Scene` class. It stores rendering parameters that the renderer will later need, such as the current background color, camera, or the point cloud to be rendered. A responsibility of the `Scene` class is to scale the point cloud so that it fits within the camera view frustum, on first load. Further scaling, rotation and translation can be later applied by the user via the GUI.

The camera class (3.6) contains the typical camera parameters, and provides methods for moving and orienting the camera. These methods are called by the `OSWindow` whenever the appropriate mouse and keyboard events are fired.

3.1.4 Rendering

For rendering, OpenGL 4.5 is used. According to the specification ([6]), OpenGL is an API that allows the programmer to define shader programs and the data used by them. The shaders themselves use the data to process geometry, rasterize it into pixels, and finally to

```
class Camera
{
private:
        glm::vec3 translation{-2.2f, 1.65f, 2.2f};
        float yaw = -45.0f;
        float pitch = -30.0f;
        float fov = 45.0f;
        float nearPlane = 0.1f;
        float farPlane = 100.0f;
public:
        glm::mat4 getViewMatrix() const;
        glm::mat4 getProjectionMatrix() const;
        void move(glm::vec3 amount);
        void rotate(glm::vec2 amount);
        void drawUI();
};
```

Figure 3.6: The camera class

calculate lighting and shading for these pixels. The end result is rendered into a framebuffer. Every OpenGL application needs to create a so called "OpenGL context" and associate it to an OS Window with a framebuffer for drawing. An OpenGL context holds information about the state of OpenGL (such as currently bound shader, or vertex data) as seen by the application. Each application using OpenGL needs a separate context. In LPCRenderer, context creation is handled alongised window creation by `OSWindow`.

Rendering is split into three parts: the currently active point cloud renderer (`PCRenderer`), the `MainRenderer` component, and UI rendering (discussed in 3.1.5).

```
class PCRenderer
{
protected:
        mutable PointCloud const* cloud = nullptr;
public:
        virtual void update() = 0;
        virtual void render(Scene const* scene);
        virtual void drawUI();
        ...
};
```

Figure 3.7: Point cloud renderer abstract base class

`PCRenderer` is an abstract base class for rendering a single point cloud and nothing else. The derived classes are where the different compression algorithms are implemented. They are explored in more detail in 3.3. Listing 3.7 shows the interface of this class. It holds a pointer to the currently rendering point cloud, and updates it each time from the `Scene` passed in through the `render` method. The `update` method converts the current point cloud to a compressed representation and streams it to the gpu, preparing it for rendering. GPU memory allocations are also tracked along the way.

```
1 void MainRenderer::render(Scene* scene)
2 {
3         glClearColor(scene->getBackgroundColor());
4         if(drawBricks)
5         {
6                 ...
7         }
8         pointCloudRenderer->render(scene);
9 }
```

Figure 3.8: The simplified rendering loop

The main renderer has a very simple public interface, providing just a function for rendering a scene, and another one for drawing its UI. Internally, the main renderer also has an instance of a `PCRenderer` called `pointCloudRenderer`. Listing 3.8 shows a simplified version of the main rendering function. After clearing the background color and conditionally drawing the brick outlines for the point cloud, the `render` method of the current point cloud renderer is invoked.

To summarise the relationship between the components, there is one main renderer that renders a single scene, selected from the scene manager. Each scene in itself, selects one point cloud from the point cloud manager to pass on to the renderer. New point clouds can be added either by importing a point cloud, or by creating a decimated version of an existing one.

3.1.5 User interface

The GUI is written with the help of the "ImGui" library, which stands for "Immediate mode GUI". In immediate mode GUI libraries (in contrast to retained mode libraries, such as Qt), the UI elements are "built" during the frame, and drawn at the end of the frame. UI elements

can be added at any point within the frame, from any part of the code. This allows for great flexibility and iteration speed when designing the UI. Things such as conditionally displaying a UI element also become easier, since we can choose to simply not build that element if a condition is not met, we do not have to remember to "hide" the element and we do not need to keep an object referring to said element.

In LPCRenderer, objects that need to display some sort of UI have a `drawUI()` method, that builds the necessary UI elements and handles updates received from the UI. Listing 3.9 shows a portion of the `PointCloud::drawUI()` member function. For reference, 3.15 shows the end result. `ImGui::InputInt` is a function that, behind the scenes, creates a widget for modyfing the value of an integer referenced through a pointer. Additionally, vertex and texture data is created for the newly constructed widget, and stored in the ImGui namespace for rendering later. Similarly, `ImGui::Button` builds a button widget and returns a boolean telling us wether the button has been pressed this frame.

Referring back to 3.1, once the main renderer is done, a free function, `drawUI` is called, that builds the menu bar, the currently visible windows, and recursively calls all the necessary `drawUI` functions to build the rest of the UI elements. Finally, the whole UI data that we built this frame is rendered.

```
1 static int decimatePointCount = 100 000;
2 ImGui::InputInt("Decimate␣Max␣Points:␣", &decimatePointCount);
3 if(ImGui::Button("Decimate"))
4 {
5         ...//decimate(decimatePointCount)
6 }
```

Figure 3.9: ImGui example

3.2 User manual

On first start-up of the program, not all windows may be visible. Hidden windows may be shown via the *View* menu in the top-left corner of the screen, on the menu bar. Point clouds can be imported via drag-and-drop. One or more `ply` and/or `conf` files can be dropped in and imported at once. They will all be merged into one point cloud instance, within the software. A typical view of the program, after importing the stanford bunny and tweaking some parameters, can be seen in 3.10.

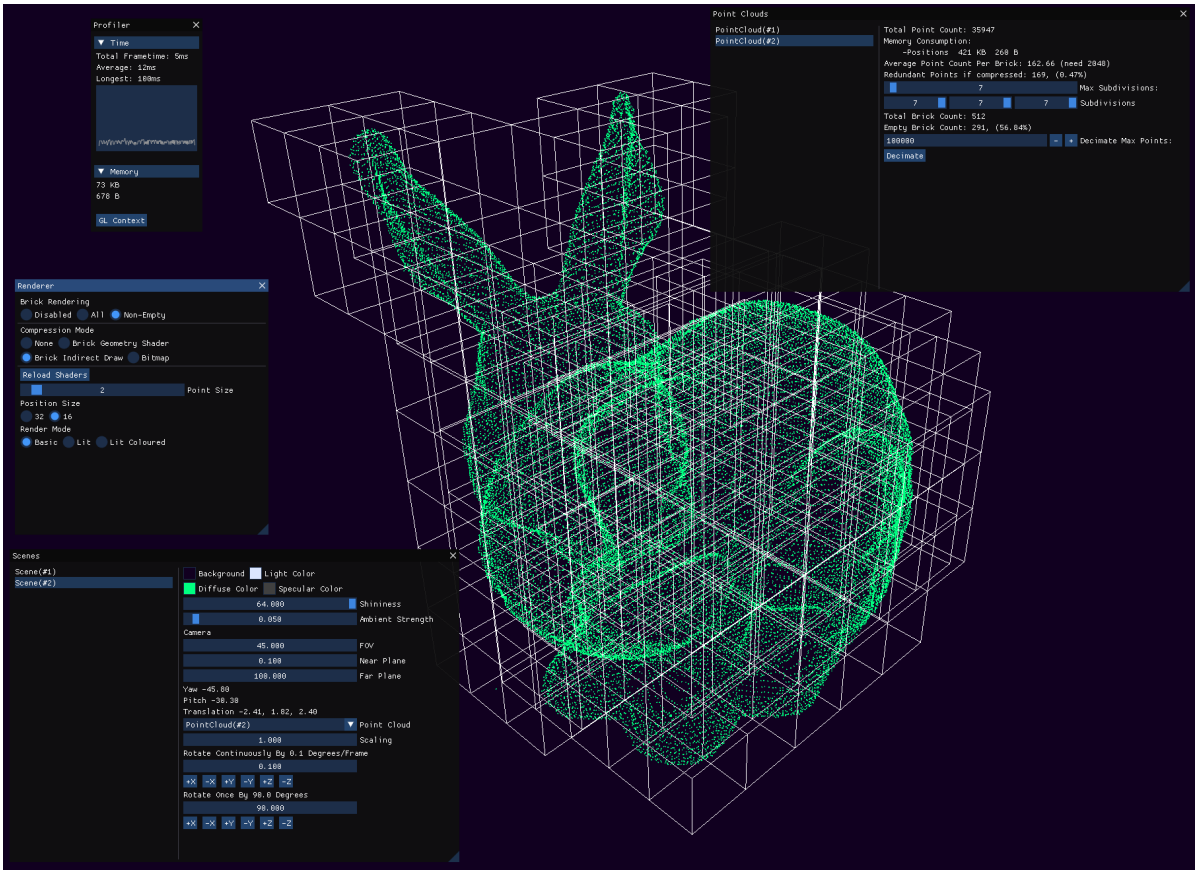

Figure 3.10: LPCRenderer with a model loaded, and all windows opened

3.2.1 Profiler window

Figure 3.11: The profiler

The profiler window (shown in figure 3.11) provides a quick overview of performance and memory consumption. The last 100 frames are sampled and the average, the current, as well as the longest duration are displayed, alongside a small plot for quickly spotting frame inconsistencies and lag spikes. Under the *Memory* header, the GPU memory in use by the current point cloud renderer is shown.

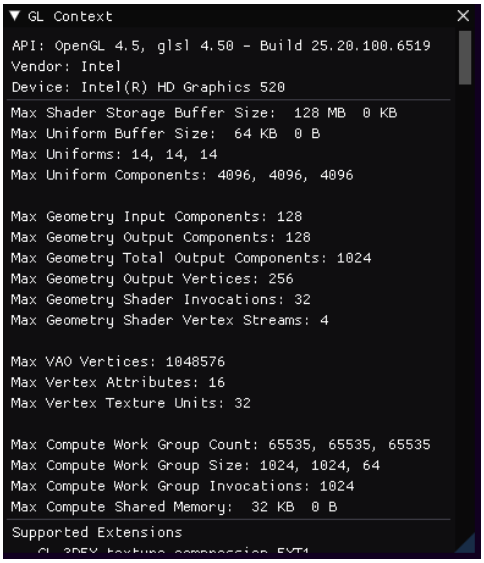

Figure 3.12: Window showing information queried about the current OpenGL context

The *GL Context* button brings up information about the OpenGL context (3.12). This includes information about the current GPU being used, supported extensions and API versions, as

well as limitations relating to geometry shader invocations, for example.

3.2.2 Renderer window

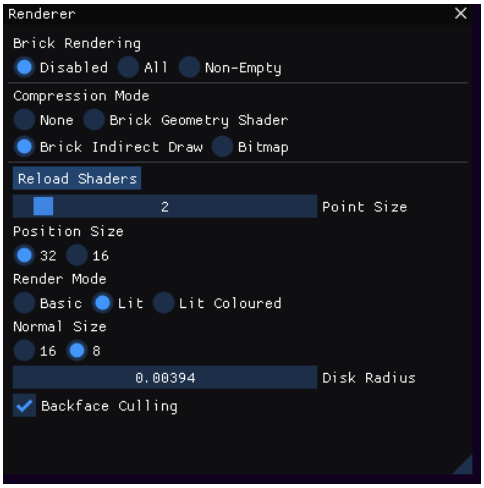

Figure 3.13: Renderer settings

The main renderer window (3.13) has two rows of radio buttons. The first row controls whether point cloud bricks are rendered or not. It is advisable to disable brick rendering when testing the performance of different point cloud renderers, and only use it to visualise where non-empty bricks are situated. The second row chooses a compression mode, and thus, a different point cloud renderer. The UI for the currently selected renderer is then displayed inline, further down. Each renderer has its own self-explanatory UI, their implementation and parameters are described in section 3.3.

3.2.3 Scene window

The scene window (3.14) is an example of a resource manager UI. A list of available resources is displayed on the left and the UI of the currently selected resource is displayed inline, on the right side. In this case, some color widgets are available. In order to change a color, the color itself needs to be clicked, bringing up a color picking window. In the middle, a combo box allows one to change the currently active point cloud for this scene.

Figure 3.14: The scene manager

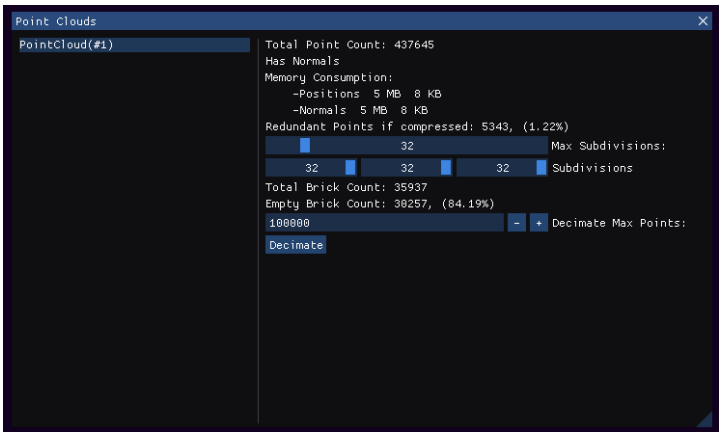

Figure 3.15: The point cloud manager

3.2.4 Point cloud window

The point cloud manager UI (3.15) displays information about the selected point cloud, pertaining to memory consumption by uncompressed positions, normals, or colors(if available). The *"Redundant points if compressed"* field shows what percentage of points are lost with the current combination of compression method/parameters and brick subdivisions. The *Max Subdivisions* slider is only there so that there is an upper limit when changing the subdivion counts below. The subdivions are updated in real-time when moving the sliders, so accidentally moving a slider to a large value could cause a lag spike, as the point cloud needs to regenerate its bricks. The ability to create a copy of the current point cloud with a specified maximum number of points is also available via the *Decimate* button.

3.3 Renderers

As discussed in 3.1.4, the `PCRenderer` class is the base class of all point cloud renderers implemented. Of these, there are four: `PCRendererUncompressed`, `PCRendererBrickGS`, `PCRendererBrickIndirect`, and `PCRendererBitmap`. Additionally, in the implementation of these renderers, shader programs have been used. A shader is a program that runs in parallel on the GPU([6]). The following shader types have been used while implementing these renderers:

- vertex shader: runs for each vertex passed on to the graphics card. Reads in vertex attributes, such as position, normal and color, and passes them on to other shader stages. Is typically used for transforming vertices from model space to screen space using the model, view, and projection matrices.

- geometry shader: is an optional shader that directly follows the vertex shader. It takes as input the outputs of one or more vertex shaders, and can generate and output n vertices. This stage is typically used for generating geometry on the GPU, such as quads from single vertices, for example.

- fragment shader: is ran for each pixel covered by the geometry resulting from the vertex and/or geometry shaders. Since multiple vertices can be responsible for the invocation of a fragment shader (for example, a fragment shader invoked inside of a filled triangle), the outputs of the provoking vertices are interpolated using barycentric interpolation before being passed on to the fragment shader. This is the last programmable shader

stage among the shaders used for rendering. It is commonly used for shading, as its output is the final color for the respective pixel.

- compute shader: the aforementioned shaders are executed together in a sequence as part of the rendering pipeline. In contrast, compute shaders, run independently of other shader types, and have no predefined inputs. It is up to the programmer to define the usage for this shader type, but, as the name implies, it is mostly used for computing. Things such as GPU particle simulation or image processing are typically implemented using compute shaders.

3.3.1 Uncompressed

`PCRendererUncompressed` is used as the baseline implementation. It uses 3 x 32bit floating point numbers for positions, and normals, if available, for a total of 192 bits per vertex, for shaded rendering without colors. For position-only rendering, the vertex is simply transformed to screen space and colored with a predefined color. If normals are available, and lit rendering is chosen, then a geometry shader is used that runs for each point in the dataset. Using the normal and the position, 4 new vertices are generated, that lie on the plane perpendicular to the normal. These 4 vertices represent the corners of an oriented quad. The distance of each vertex to the center of the quad is sent as an output when the vertex is emitted. Due to interpolation, each fragment shader invocation for the resulting quad knows where the current pixel lies on the quad surface. If the pixel lies outside of a specified radius, it is discarded, thus leaving the shape of an oriented disk behind. The fragments that are inside the disk are lit using blinn-phong shading.

3.3.2 Brick geometry shader

The first attempt at implementing the brick-relative vertex compression method involves geometry shaders. The idea is to invoke the vertex shader once per brick, and then use the geometry shader to generate the vertices belonging to that brick. We do this by putting all packed positions into a single buffer, and then indexing into it from each brick, as seen in diagram 3.18.

Figure 3.16 shows the vertex shader for this implementation. The offset into the positions buffer, as well as the number of positions belonging to this brick are received through vertex attributes and sent further to the geometry shader for processing. The variable `gl_Position`

representing the world position of the current brick is calculated using the brick index, brick size, and cloud origin. The geometry shader will access this variable later and use it as an offset to calculate the world position of each owned vertex.

```
1  uniform vec3 cloudOrigin;
2  uniform vec3 brickSize;
3  layout(location = 0) in uvec3 brickIndex;
4  layout(location = 1) in uint bufferOffset;
5  layout(location = 2) in uint bufferLength;
6  out VS_OUT
7  {
8          uint bufferOffset;
9          uint bufferLength;
10 } vs_out;
11 void main()
12 {
13         vs_out.bufferOffset = bufferOffset;
14         vs_out.bufferLength = bufferLength;
15         gl_Position = vec4(cloudOrigin + brickIndex * brickSize, 1);
16 }
```

Figure 3.16: Vertex shader for PCRendererBrickGS

In figure 3.17, the geometry shader itself can be seen. The idea here is to run through all the positions that belong to the current brick, unpack them, and emit them. This, however is where we start running into issues.

OpenGL imposes certain limitations on the geometry shader stage ([6], [5]), relating to the maximum number of vertices emitted by a single geometry shader invocation and maximum amount of memory occupied by the output of said vertices. On the used hardware (Intel HD520), the maximum number of vertices emitted is 256. This number can be pushed a bit further by using so called "instanced geometry shaders". This effectively allows us to launch up to 32 separate, independent geometry shader instances per provoking vertex. Thus, the theoretical maximum number of vertices per brick is $32 \cdot 256 = 8'192$. However, even this theoretical maximum cannot be achieved, as other limits are hit even sooner. There is a limit on the maximum number of components emitted per geometry shader invocation as well: 128 components. This limit includes the built-in `gl_Position` components. So it would seem we can only emit $\frac{128}{3} = 42$ vertices per geometry shader instance, or $42 \cdot 32 = 1'344$ vertices per brick. If we wish to draw oriented disks, the number falls further down to 336, as we would need to emit four times as many vertices.

```
1  #define MAX_VERTS 256
2  layout(points, invocations = 32) in;
3  layout(points, max_vertices = MAX_VERTS) out;
4  layout(std430, binding = 0) buffer PositionsBuffer
5  {
6          uint positions[];
7  };
8  uniform vec3 brickSize;
9  uniform mat4 model;
10 uniform mat4 view;
11 uniform mat4 projection;
12 in VS_OUT
13 {
14         uint bufferOffset;
15         uint bufferLength;
16 } gs_in[];
17 void main()
18 {
19         for(int i = 0; i < MAX_VERTS; i++)
20         {
21                 uint index = gl_InvocationID * MAX_VERTS + i;
22                 if(index >= gs_in[0].bufferLength)
23                         return;
24                 index += gs_in[0].bufferOffset;
25                 vec3 position = unpackUnorm4x8(positions[index]).xyz;
26                 position *= brickSize;
27                 gl_Position =
28                         projection * view * model *
29                         vec4(gl_in[0].gl_Position.xyz + position, 1.0f);
30                 EmitVertex();
31                 EndPrimitive();
32         }
33 }
```

Figure 3.17: Geometry shader for PCRendererBrickGS

As can be seen in Sascha Willems' online gpu capabilities database ([7]) even very high end graphics chips do not fare much better. The highest theoretical maximum is hit by the AMD Radeon RX Vega chipset. With a maximum of 1'024 vertices emitted per geometry shader invocation, 127 maximum geometry shader instances per provoking vertex and 16,384 maximum emitted components per geometry shader invocation, an RX Vega can render bricks with up to 130'000 vertices per brick if rendering only positions, or 32'000 vertices if rendering oriented disks. This number seems reasonable, however there is one additional caveat: geometry shaders get very slow when emitting a large number of vertices. This is because the OpenGL specification requires that all geometry shader invocations provoked by a vertex write their output in order ([6]). This means that execution cannot be easily parallelised, and performance suffers greatly.

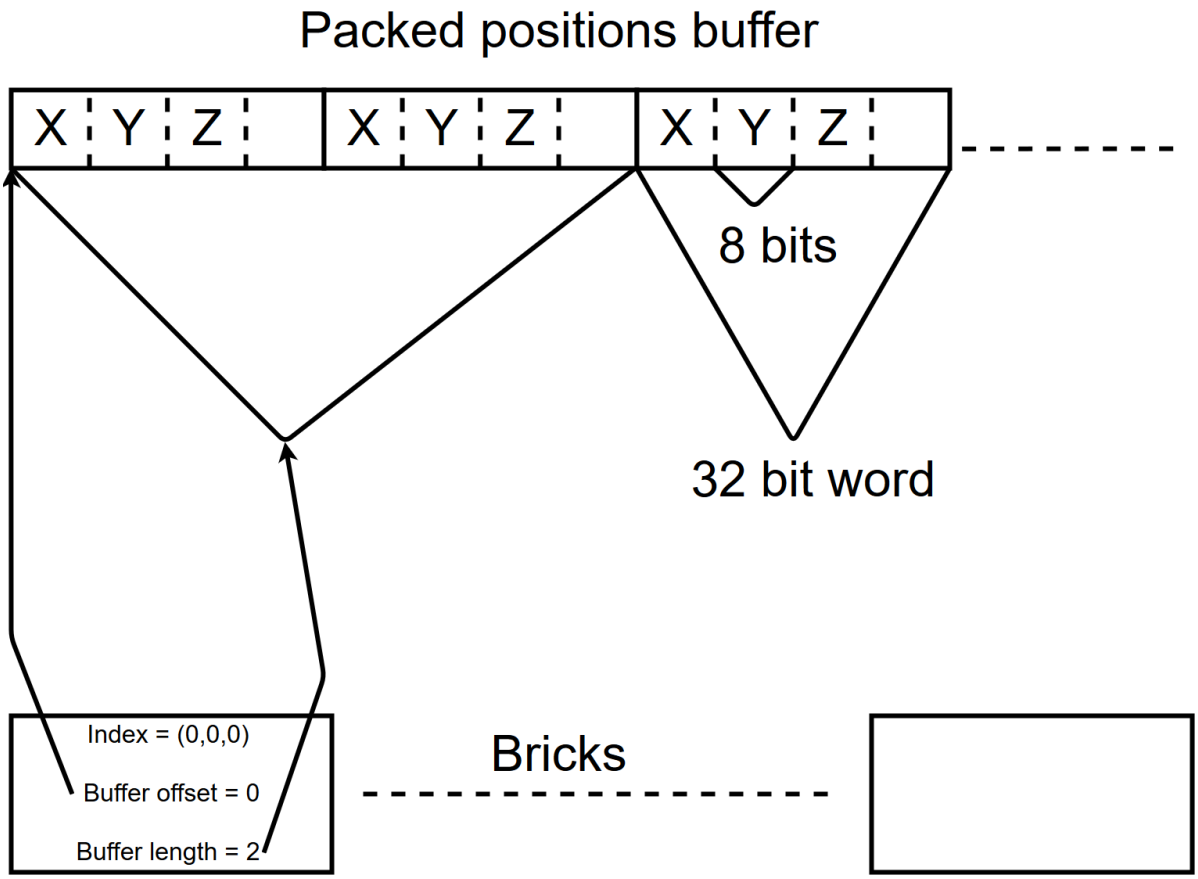

Figure 3.18: Data layout for PCRendererBrickGS

3.3.3 Brick indirect

The reason we tried to use geometry shaders in the previous attempt was because we need a way to tell a vertex shader which brick it belongs to. Vertex attribute divisors ([6]) enable the programmer to send each element in an attribute buffer to N vertex shaders, instead of just to one. The problem however is, that the number of vertices per brick is not fixed, and thus this method cannot be used. Another "solution" would be to issue a draw call from the cpu, for each brick, and just change a shader uniform that specifies the index of the current brick. While functional, this method would incur large driver overhead, since each command would have to pass from the program, to the driver, and finally down the PCIe lane to the GPU. Driver overhead can very quickly become a bottleneck, and more modern graphics API such as Vulkan or DirectX12 strive to reduce this overhead as much as possible.

Luckily, in OpenGL 4.5 it is possible to issue so called "indirect draw commands". Figure 3.21 demonstrates this concept. A special buffer used only for defining draw commands is allocated on the GPU. Each draw command is a struct with four unsigned ints (so 128 bits per draw command). The members `first` and `count` specify a range into another GPU buffer from which the corresponding draw call should create vertex shader invocations. The member `instanceCount` is present, since instanced rendering can be used with this technique, in our case this parameter is always 1. The `baseInstance` member has a purpose when doing instanced rendering, but is unused in other cases. This means we can use it for storing the index of the brick corresponding to that draw call. Most importantly, this variable can be read from within the shader itself, which means we have found our solution. In this way, it is as if we would be doing multiple draw calls in a loop, from the CPU, updating the brick index each time, but everything takes place on the GPU, with just a single API call taking place on the CPU (`glMultiDrawArraysIndirect`).

Figure 3.19 shows how the vertex shader looks like when rendering only positions for this method. The renderer has the option to send positions packed into 32- or 16-bit integers (with 10, respectively 5 bits of precision available for each axis).

If normals are available, a different version of the same shader is used, that also does normal decoding. This can be seen in figure 3.20. The normals are sent as spherical coordinates with a choice between 32- and 16- bit integers (16 respectively 8 bits of precision per angle). Packing the normals into 16 bit integers seems to deliver almost no perceptible loss in visual quality.

```
uniform vec3 cloudOrigin;
uniform vec3 brickSize;
uniform uvec3 subdivisions;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform int positionSize;
layout(location = 0) in uint compressedPosition;
void main()
{
	vec3 relativePosition;
	if(positionSize == 16)
	{
		relativePosition.x =
			float(bitfieldExtract(compressedPosition, 0, 5)) / 32.0f;
		relativePosition.y =
			float(bitfieldExtract(compressedPosition, 5, 5)) / 32.0f;
		relativePosition.z =
			float(bitfieldExtract(compressedPosition, 10, 5)) / 32.0f;
	}
	else if (positionSize == 32)
	{
		relativePosition.x =
			float(bitfieldExtract(compressedPosition, 0, 10)) / 1024.0f;
		relativePosition.y =
			float(bitfieldExtract(compressedPosition, 10, 10)) / 1024.0f;
		relativePosition.z =
			float(bitfieldExtract(compressedPosition, 20, 10)) / 1024.0f;
	}
	relativePosition *= brickSize;
	uint index = gl_BaseInstanceARB;
	uvec3 indices;
	indices.z = index / ((subdivisions.x + 1) * (subdivisions.y + 1));//count surfaces
	index = index % ((subdivisions.x + 1) * (subdivisions.y + 1));
	indices.y = index / (subdivisions.x + 1);//count lines
	indices.x = index % (subdivisions.x + 1);//count points
	vec3 brickOrigin = indices * brickSize;
	gl_Position =
		projection * view * model *
		vec4(cloudOrigin + brickOrigin + relativePosition, 1);
}
```

Figure 3.19: Vertex shader for PCRendererBrickIndirect

```
layout(location = 1) in uint compressedNormal;
vec3 decodeNormal()
{
        vec2 s;
        s.x = float(bitfieldExtract(compressedNormal, 0, normalSize))
               / (1 << normalSize);
        s.y = float(bitfieldExtract(compressedNormal, normalSize, normalSize))
               / (1 << normalSize);
        float theta = s.y * pi;
        float phi = (s.x * (2.0 * pi) - pi);
        float sintheta = sin(theta);
        return vec3(sintheta * sin(phi), cos(theta), sintheta * cos(phi));
}
```

Figure 3.20: Vertex shader normal decoding for PCRendererBrickIndirect

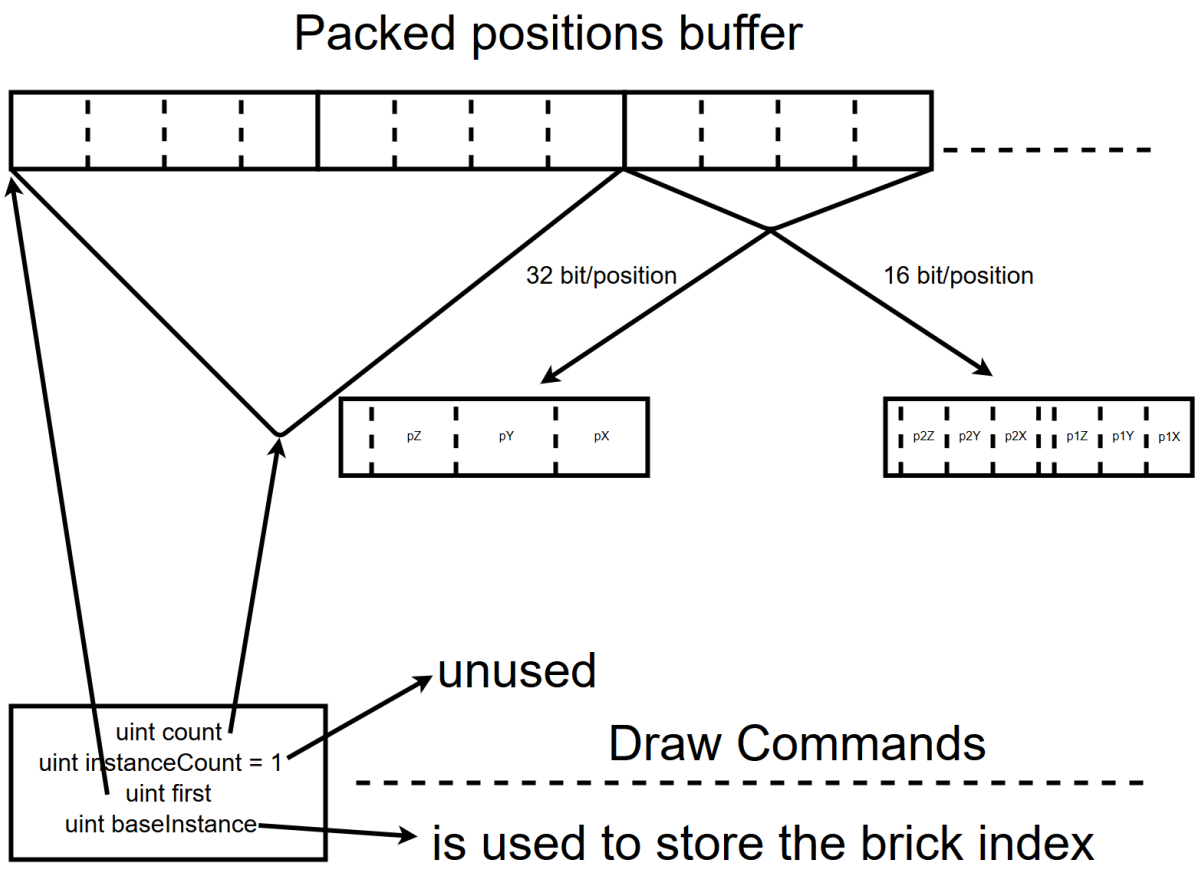

Figure 3.21: Data layout for PCRendererBrickIndirect

3.3.4 Bitmap

The last method converts the point cloud into bitmaps before sending them to the GPU to be unpacked and rendered. The process is accomplished in two steps, as diagram 3.22 shows.

We store all non-empty bitmaps on the GPU in a large buffer. Since we are not storing all bitmaps, we cannot know the index of the bitmap brick, from the position of the bitmap in the buffer alone, thus the brick indices are stored in a separate buffer. Using a compute shader, the bitmaps need to be unpacked into separate buffers, and then rendered in another pass. A "staging area" is prepared for this purpose, in the form of a draw command buffer of fixed size N. Since the draw commands will need to reference an existing positions buffer, a buffer with enough space to hold the maximum number of possible positions in a bitmap is reserved for *each* of the N reserved draw commands.

From the CPU side, we know we have M bitmaps that need to be rendered. We choose a batch size N, and, in a `for` loop, unpack N bitmaps at a time, and fill the preallocated command and positions buffers using a compute shader. Then, in a sperate API call, we render the current batch using the `PCRendererBrickIndirect` method. We continue until there are no more bitmaps left to render, thus finishing the rendering of [one] frame.

Sadly, this method incurs quite a lot of driver overhead when using a batch size of $N = 1$. Choosing an appropriate batch size is a balancing act however, since large batch sizes need to reserve ever larger positions buffer. Chapter 4 goes more in depth into some benchmarks relating to this issue.

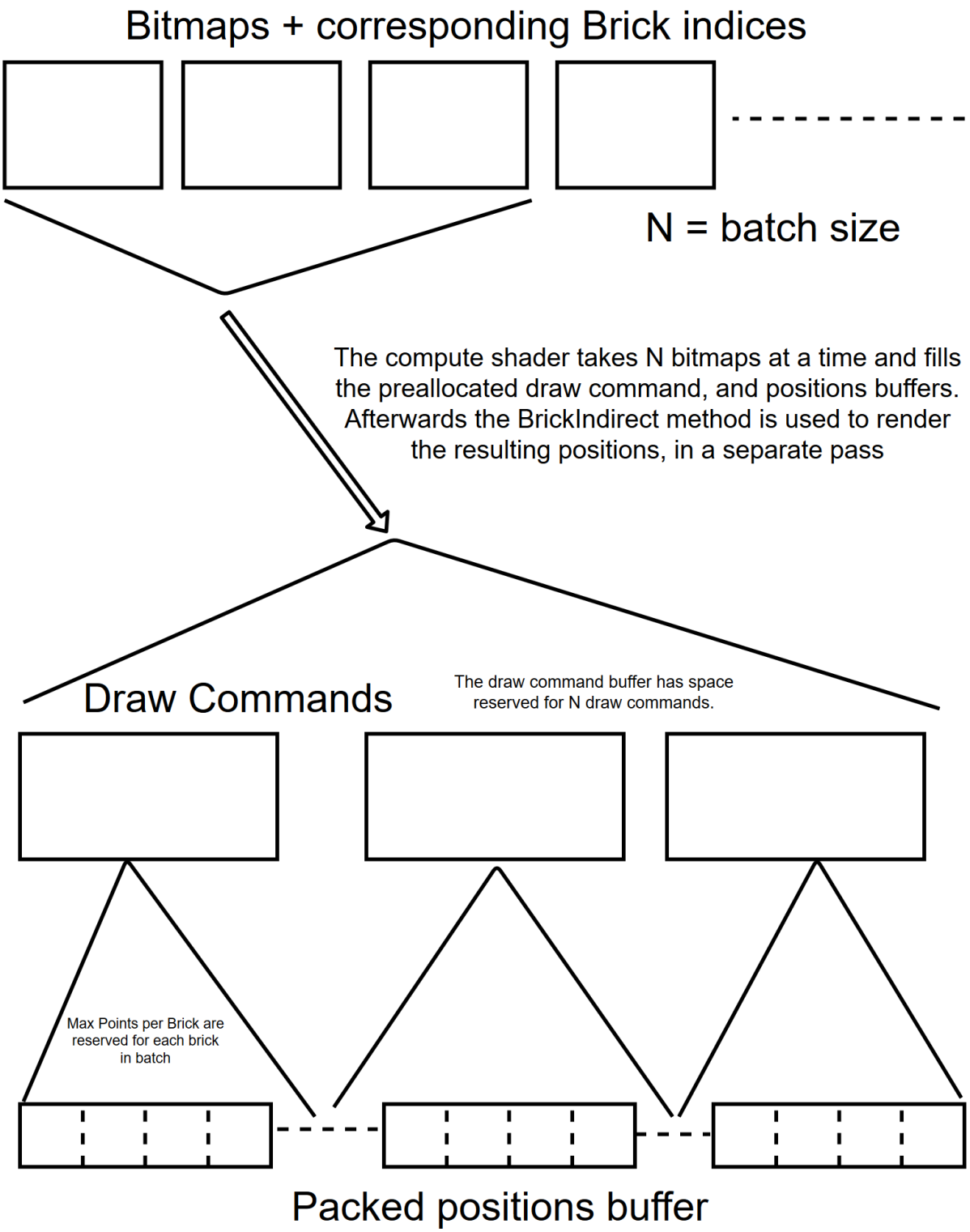

Figure 3.22: Data layout for PCRendererBitmap

4 Results

Table 4.1: Bitmap sizes

Size	Max points	Memory	Bits/Position	Packed position size	Points threshold
512	134'217'728	16MB	27	32	4'194'304
256	16'777'216	2MB	24	32	524'288
128	2'097'152	256KB	21	32	65'536
64	262'144	32KB	18	32	8'192
32	32'768	4KB	15	16	2'048
16	4'096	512B	12	16	256
8	512	64B	9	16	32
4	64	8B	6	8	8

Table 4.1 shows the relationship between bitmap size and memory consumption, compared to storing the relative offsets directly. The "Bits/Position" column shows us how many bits per point we would need if we were to store the point offsets directly. Because of memory alignment, the actual memory per point would be the one in the "Packed position size" column. Finally, the "Points threshold column calculates how many points would need to be stored inside the bitmap, in order for the bitmap method to be more memory efficient than the brick indirect method.

It would seem in this case, that smaller bitmap sizes are preferable, since less points are needed per brick in order to make the compression scheme worthwile. However, with smaller bitmap size comes a loss in precision, and a higher chance for multiple points to be represented by the same bitmap bit. We can regain precision, however by increasing the number of brick subdivisions overall. The relationship between these different factors is illustrated in the following charts.

The different methods are shown at the bottom of the charts, starting with the baseline, uncompressed renderer. Next up is the BrickIndirect renderer with positions packed into a 32- or 16-bit integer. The rest are the various "BitmapN B" variations wih N being the size of one side of the bitmap (N * N * N bits in total) and B, the batch size used. For each method,

the blue column shows the time it took to render a frame, averaged over 100 samples. The green column shows the memory consumption, and the orange points show what percentage of points have been lost due to insufficient precision, with 0% being at the very bottom of the chart, and 100% at the top. Each chart description tells us how many bricks were used. For example 4.1 used 8 x 8 x 8 bricks.

Starting off with chart 4.1, we can see that the BrickIndirect32 method is superior in memory consumption to all the other, getting very close to baseline in performance, and with virtually no information lost. All the other methods prove to be unusable at this brick count, since they lose too many points.

Case 4.2 proves not much better. Some bitmap methods start retaining more information now, although still not a usable amount. Chart 4.3 draws almost the same conclusion. What is different this time, is that the BrickIndirect32 method starts getting slower. In fact, all methods, except baseline start slowing down, due to the fact that there are so many individual bricks to be drawn.

Figure 4.4 is the most interesting, since this is the first instance where a bitmap method starts preserving enough information to be usable, namely Bitmap32 only loses 3% of its points. However, memory consumption is higher than baseline, due to the fact that a large batch size has to be used, and performance is still very bad. In this case however, BrickIndirect16 might be considered, since it uses only half the memory of BrickIndirect32, at the same (bad) performance.

It is clear from these benchmarks, that the bitmap method of encoding point cloud positions is not usable for point clouds that represent surfaces. In all cases, the BrickIndirect method is faster, more memory efficient, or both, while retaining almost 100 % precision.

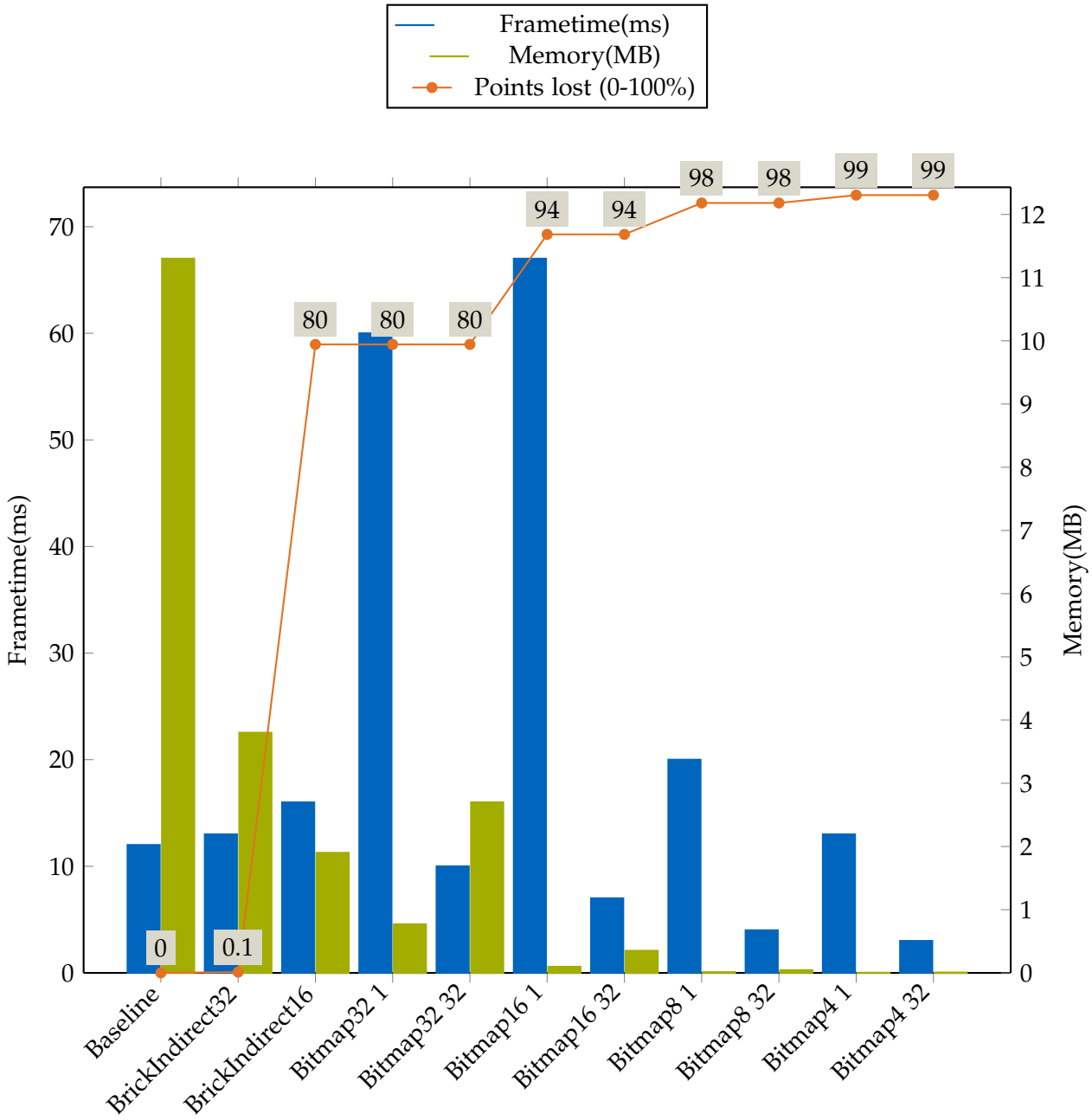

Figure 4.1: Neuschwanstein 8x8x8 @1080p Intel HD520

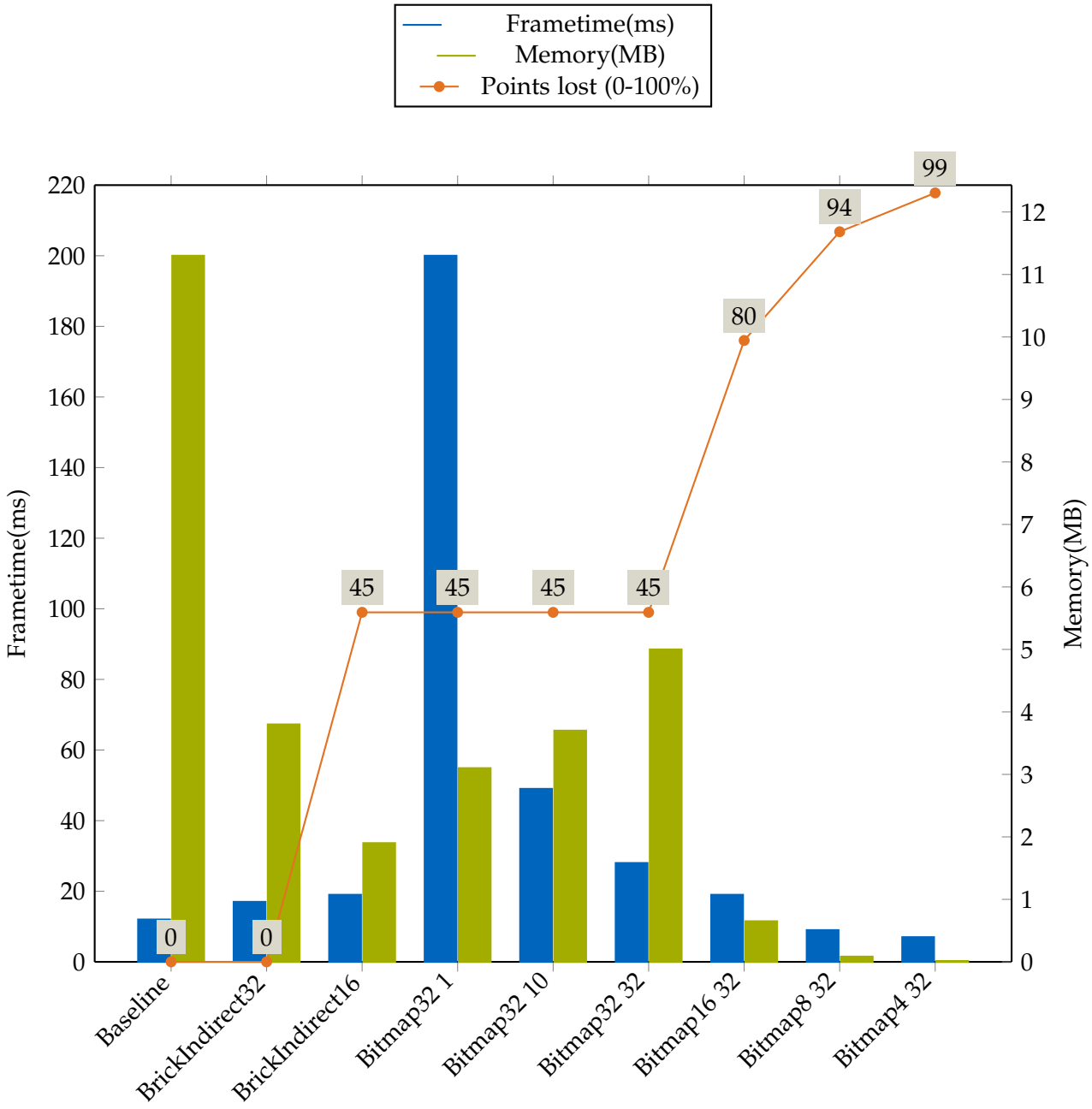

Figure 4.2: Neuschwanstein 16x16x16 @1080p Intel HD520

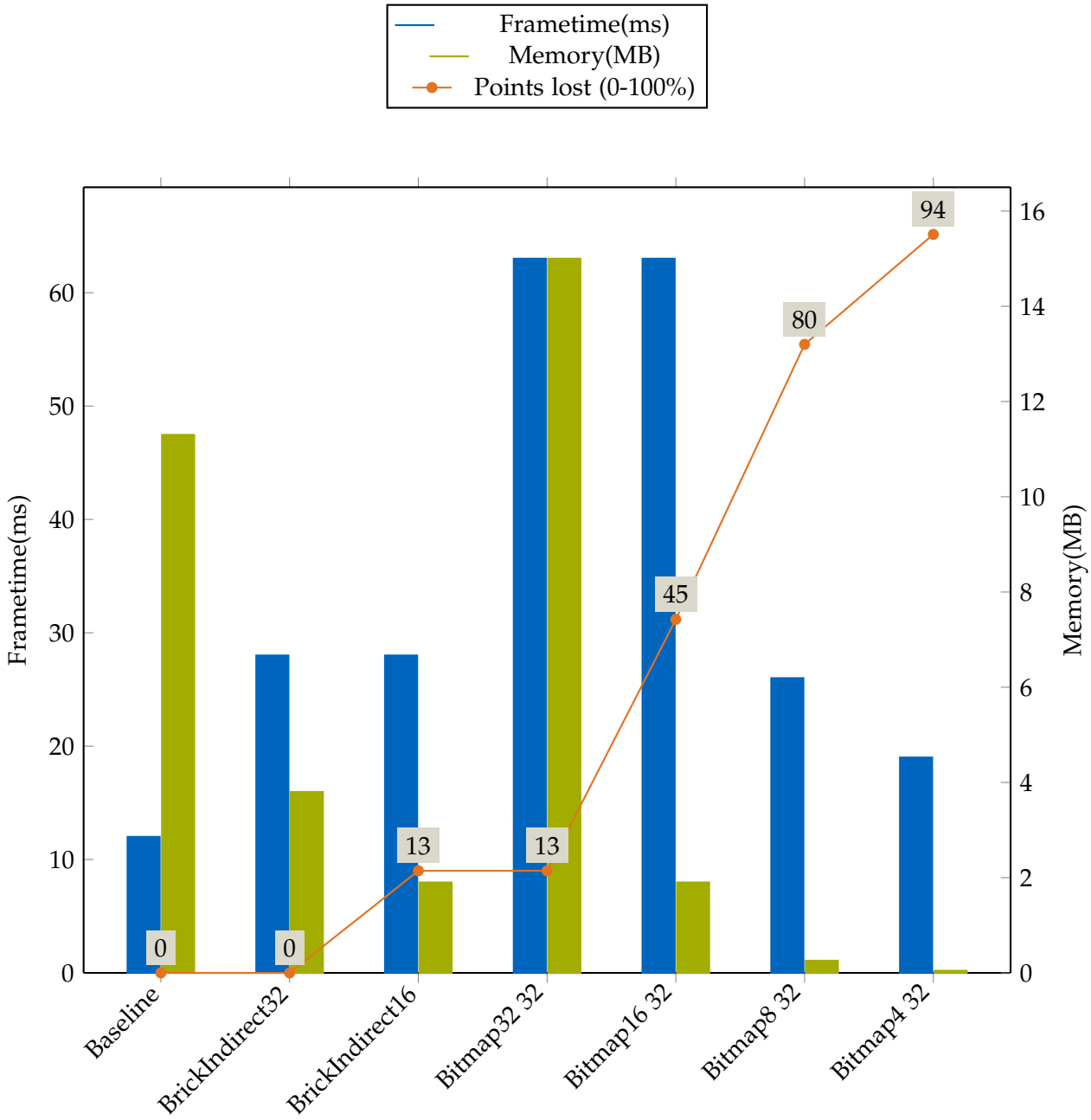

Figure 4.3: Neuschwanstein 32x32x32 @1080p Intel HD520

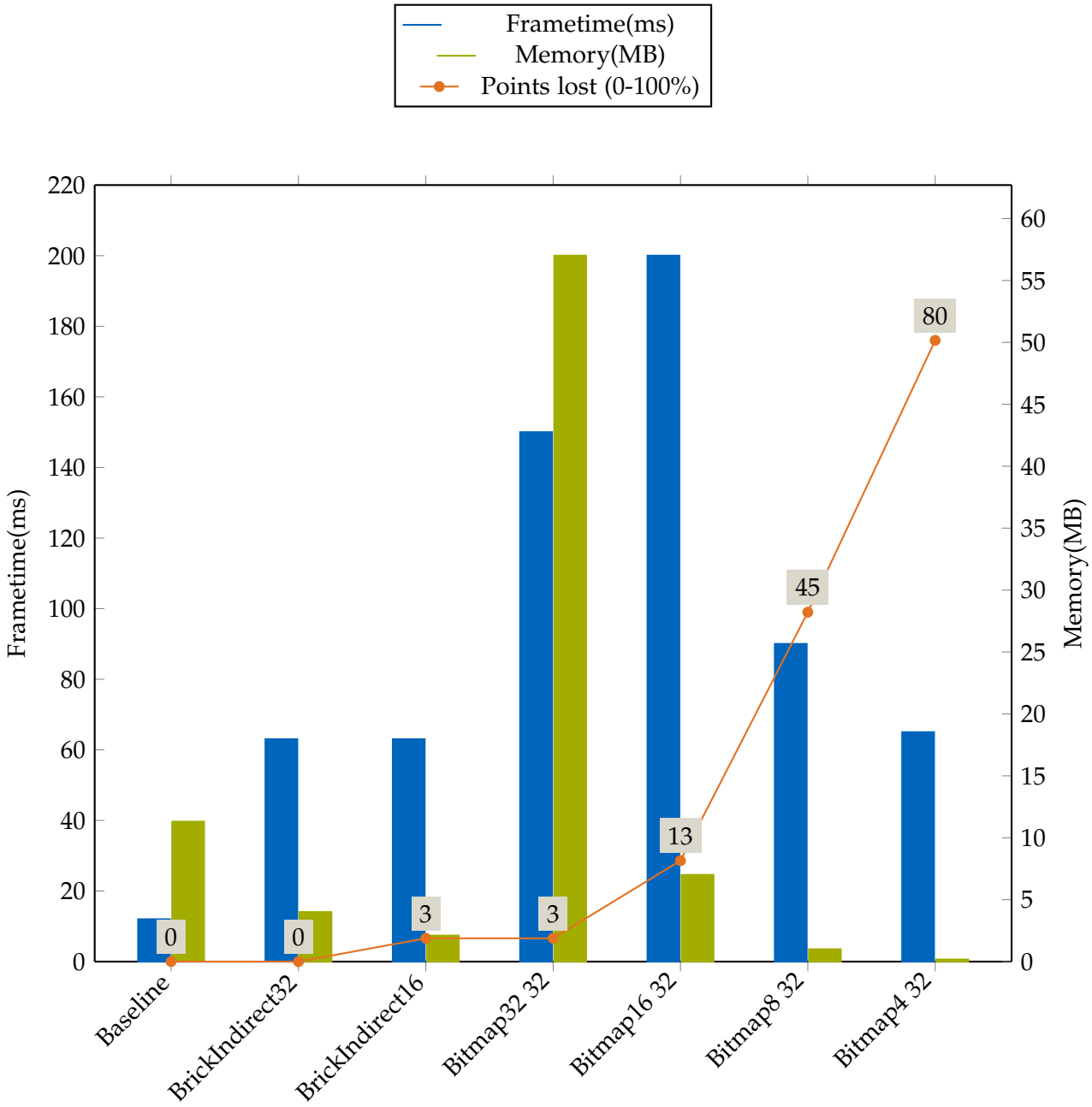

Figure 4.4: Neuschwanstein 64x64x64 @1080p Intel HD520

5 Future work

Although the bitmap method has proved to be impractical, there are areas where the BrickIndirect method might be improved further, in regards to performance.

Frustum culling could be implemented in a compute shader, in order to avoid processing bricks that are not even visible. One compute shader instance per brick could cull the bounding box of the brick against the view frustum, and set the "count" field of the corresponding draw command to 0 if the brick is deemed hidden.

Point cloud LOD could be implemented, based on the surface area a brick takes up on screen. Bricks with very small surface area, where detail can hardly be distinguished anyway, could render only a portion of their points, in order to save on rendering time.

Occlusion queries could be used by drawing the bricks front-to-back in slices. After each slice, a "dummy" brick would be drawn first, in the form of a solid cube. Using an occlusion query on this cube, we would find out if any of its fragments are at all visible. If the brick is completely occluded by already rendered bricks, then we save ourselves some rendering time. However due to the performance cost a round trip to the CPU causes, one would have to use the results of occlusion queries from previous frames.

6 Conclusion

In conclusion, the BrickIndirect method of compressing point clouds proves to be better than the bitmap method. With the techniques described in the Future Work chapter, one could further enhance the BrickIndirect method and write a renderer that is superior to the baseline method in both performance and memory.

List of Figures

Bibliography

[1] J. F. Blinn. "Models of light reflection for computer synthesized pictures". In: *ACM SIGGRAPH computer graphics*. Vol. 11. 2. ACM. 1977, pp. 192–198.

[2] Z. H. Cigolle, S. Donow, D. Evangelakos, M. Mara, M. McGuire, and Q. Meyer. "A survey of efficient representations for independent unit vectors". In: *Journal of Computer Graphics Techniques* 3.2 (2014).

[3] K. McHenry and P. Bajcsy. "An overview of 3d data content, file formats and viewers". In: *National Center for Supercomputing Applications* 1205 (2008), p. 22.

[4] M. Levoy, J. Gerth, B. Curless, and K. Pull. "The Stanford 3D scanning repository". In: *URL http://www-graphics. stanford. edu/data/3dscanrep* 6 (2005).

[5] M. Segal and K. Akeley. *The OpenGL® Shading Language*. The Khronos Group Inc. 2017.

[6] R. R. John Kessenich Dave Baldwin. *The OpenGL® Graphics System: A Specification (Version 4.5 (Core Profile)*. The Khronos Group Inc. 2015.

[7] S. Willems. *Vulkan Hardware Database*. URL: `http://vulkan.gpuinfo.org/`.