A 3D Graphics Library For Resource Limited Situations

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Bachelor of Computer Science with Honours
School of Computing and Communications

Lancaster University

by

Dexter Latcham

March, 2024

Declaration

Abstract

This project will detail the design and implementation of Imej, a modular 3D graphics library developed in the C programming language. This will be created with resource limited situations in mind, allowing for a real time rendering to be created without the need for dedicated graphics hardware. Functionality to load mesh structures, manipulate them in 3D space and produce a 2D view will all be included. Through a portable design with no external dependencies, this library will provide flexibility for any use case. We include support for displaying a render through a range of mediums. Included is support for conventional uses such as a desktop window but to demonstrate the low resource requirements of Imej, a driver for the BBC MicrobitV2 board is provided.

Contents

1 Introduction

The field of computer graphics benefits from decades of research, from both academic institutions and corporations alike. Since the term was coined in the 1980s, continuous development has provided us with the technology to create photorealistic scenes in real time[1]. The impact computer graphics has on the modern world is difficult to underestimate. A diverse range of sectors benefit from the technology, ranging from uses in DNA sequencing[2] to architectural design. Modern hardware has become so performant that Disney has recently switched to filming against computer generated backdrops, which update in real time with the movement of the camera[3].

In the early days of computer graphics, images were produced through a fixed-function pipeline[4]. This pipeline describes the sequence of events used to transform a 3D scene to the 2D image. The processes involved in computer graphics are often highly parallel operations[5]. As such, Graphical Processing Units were developed specifically to perform these operations in parallel at high speeds. These ASIC devices incorporate hardware-level optimisations to provide the best performance. The power offered by these devices is unparalleled, with their performance growing faster than any other microchip[6].

With these devices however comes a problem of portability. Each vendor offers unique architectures, each with their own complexities. To allow for these to be fully leveraged by a developer, a graphical API is used. These provide an abstraction between a programmer and the hardware their application is executed on. These graphical APIs are used to define a standard interface used to interact with the graphics hardware. An API can have many different implementations offered, allowing hardware vendors to provide versions tailored for their devices.

In the early 2000s a transition began to a shader-based pipeline[7]. This offered more flexibility when compared to the fixed-function predecessor, allowing for customisation at each stage in the process.

In parallel to the development of these powerful systems, there has been an increased adoption of Internet of Things (IoT) technology. These low-power and resource limited embedded systems have become integral in devices ranging from smart applications to wearable technology. As a response to this, specialised APIs have been developed for use in these constrained applications.

1.1 Objectives

In this paper, we will detail the creation of the Imej library. This provides a custom API which provides the functionality to create and render 3D scenes. This is designed for use in resource limited situations, offering functionality to create 3D renders with minimal resource requirements. To achieve this, we will create our own fixed-function graphics pipeline, designed to run on a single thread. The interface provided will be written at a slightly higher level of abstraction to mainstream APIs, with inbuilt methods handle 3D models and perform basic manipulations. The implementation provided will have a modular design, making modification and extensions trivial. Included in this will be functionality to create, manipulate and animate objects in a 3D space. A provided set of transformations will allow for models in the scene to be modified through a simple interface. As this library has a portable design, we will create several options for the presentation of a frame.

1.2 Structure

- Background: an analysis of popular graphical APIs, their uses and limitations.
- Methodology: planning the architecture Imej will use and exploring the theory it will require.
- Implementation: the process of implementing the Imej library, including any changes made to the initial design.
- Using Imej: the process of compiling and using the library.
- Evaluation: testing and benchmarking our implementation.
- Conclusion: reflections after completing work on the Imej library.

2 Background

Existing graphical APIs

Most modern graphical applications rely heavily on Application Programming Interfaces (APIs) to interact with the underlying hardware[8]. A graphical API provides a standardised means for a developer to create graphical applications. These allow a developer to interact with the underlying hardware from a high-level context. As these specifications do not include implementation details, it is up to vendors to create drivers to provide this functionality.

OpenGL

With an initial release in 1992, OpenGL was the most widely used cross platform API[9]. Whilst this was designed with the intention of enabling hardware acceleration, there exists entirely CPU bound software implementation. Whilst this API is defined in a style similar to the C programming language, its use is language independent. Language bindings exist which extend its use to programming languages such as JavaScript and Java.

Vulkan

Vulkan is the successor to OpenGL[10]. Released in 2016 by the Khronos Group, this builds on the work of OpenGL, bringing developers more control over the GPU. Additionally, this brings better support for multi-core CPUs.

DirectX

DirectX is the umbrella term for a series of APIs offered by Microsoft for use on the Windows operating system[11]. In contrast to other technologies, the DirectX runtime is entirely proprietary and distributed under a closed source licence. In using a closed source licence, a developer is prohibited from examining the underlying implementation.

OpenGL ES

Graphical programming for embedded applications brings a unique set of challenges. Systems with limited resources which lack modern dedicated hardware are ill-suited for a fully featured graphics library. OpenGL exists to fill this gap. With its use cases ranging from mobile devices to wearable technology, this has become the most widely use graphical API[12].

3 Methodology

3.1 Choice of resources

3.1.1 Language

The Imej library is designed for high performance in real-time applications. To accomplish this, a low level language will be required. Direct control of memory is essential to allow us to minimise our usage, as well as avoiding the periodic halts a garbage collector can cause. Additionally, direct control of execution is essential for creating high performance code. We identified both C and C++ as potential languages which meet this criteria. C++ was strongly considered. As a superset of C, it provides the same level of control with several additions. Exception handling, polymorphism and a comprehensive standard library are all benefits offered over standard C. Despite this, we decided to use the C programming language for this library. Proficiency in the C programming language was the main factor. The C programming language will still provide us with sufficient tools to create the Imej library.

3.1.2 Valgrind

Valgrind is an open source and FOSS tool for profiling applications[13]. Through executing code on a virtual machine, Valgrind provides tools including a cache profiler, memory validator and heap profiler. In the development of Imej, we will make use of the Massif tool. This is used to track each memory allocation made whilst our software is running. Using this, we can log the total memory usage and evaluate the feasibility of running an application on resource limited hardware.
In addition, Callgrind will be used in the development of this library. This tool tracks the number of invocations made to a function, allowing for potential bottlenecks in our implementation to be identified and optimised.

3.1.3 Display mediums

The set of display mediums we include in this project are detailed in this section. These were chosen to demonstrate the versatility of the library created.

Exporting

In order to save a rendered frame to permanent storage, a suitable filetype needed to be chosen. Formats such as PNG and JPEG were considered, however, these use complex compression mechanisms to store data. For this, we require a format which could be implemented from scratch in a short time frame. The Portable PixMap format satisfies these requirements. A PPM file stores images as a sequence of pixel colour data. An image beginning with the characters ‘P3’ stores data in a human readable format. A file beginning with ‘P6’ instead stores the raw binary data of pixels in the image.

X11

To demonstrate this library running on a Linux desktop machine, we need to interact with a windowing system. Whilst desktop distributions of Linux have began migrating to the Wayland protocol, we will instead offer support for the X windowing system. X11, which has existed since 1984, is widely supported and applications written to communicate with the X server can still run on a Wayland client through the XWayland project. For these reasons, we made the decision to write the desktop driver based on the XLib system. For the use of this implementation, we will require the necessary X11 headers be installed, although, this can be assumed if the code is intended to run on a X11 machine.

Terminal

To display a render with less reliance on a graphical interface, we can instead print a low resolution image to a terminal window. This gives added flexibility as the image can either be viewed in a terminal directly connected to the machine or through a SSH session. This can be implemented using several means. For purely ASCII terminals, the character set must be ordered by surface area, allowing selection using the intensity of a pixel. If a terminal emulator which supports ANSI escape codes is being used, we can improve this visualisation. Through these escape codes, we can colour the terminal and extend our character set to simulate a higher resolution at the output.

Microbit

The Imej library is designed around low resource usage. With this property, our main use case will be for low power embedded systems. We can evaluate the effectiveness of our implementation through empirical data, including recording memory use and single threaded performance. We can improve on this through demonstrating the library being used on actual embedded hardware. For this, we will provide a visualisation being ran on the BBC MicrobitV2. This board is built around the NRF52833 chip, with a clock speed of 64MHz and 128Kb of ram. To develop code for this board, we will use the Codal runtime. This provides the functionality required for memory management as well as drivers for hardware available on the board.

3.2 Design

The architecture of the Imej library has been split into a series of modules. Each of these modules is responsible for a distinct set of functionality. The public interface these modules provide will be defined in its header file. Here we will detail how we designed these modules and the division of functionality they must provide. As part of this, we will explain concepts and theory which will be key for creating an implementation of these modules.

3.2.1 Coordinate spaces

To process 3D graphics in the Imej library, we use a three element row vector to represent a coordinate position. When a coordinate point is described in this paper, it will be within the context the coordinate space it occupies. During the rendering process, a vector will undergo a series of operations before we can find its effect on the frame. The properties and meaning of a vector at each of these stages can be found through the coordinate space it occupies. To define the rendering process, we must first define the coordinate spaces used at each step and their significance.

- Object space
  In this space, vectors are used to represent the vertices of a mesh structure. The mesh structure has an accurate representation relative to its own origin. Its location and orientation however are not defined relative to other models or the main coordinate system itself.

- World space
  In world space, the vertices of an object describe its actual location in the scene. The position, scale and rotation of an object has been set and are all accurate relative to other models in the world space.

- View space
  To create a 2D image from a scene, we define a camera with position

and orientation in the world space. The view space defines coordinates relative to this camera. The transformation which moves from world space into view space must also move the camera position to the origin and orient it to be facing the positive z axis.

- NDC space
  The NDC space is used to define points relative to the screen they will be drawn to. Coordinates which lie within a given range will later be interpolated from this into the bounds set by the display. OpenGL defines this range as x (-1 to 1), y (-1 to 1), z (-1 to 1). For the image library, we will instead define valid NDC coordinates using the range x (-1 to 1), y (-1 to 1), z (0 to 1).

- Screen space
  Once a vertex has been transformed from the NDC space to screen space, it can be mapped directly to a position on the framebuffer. This transformation entails interpolating each vertex from the range we defined in our NDC space, to the range given by the display.

3.2.2 Graphics pipeline

The term 'graphics pipeline' is used to reference the sequence of events used to produce a frame. This process is the method we use to create the 2D image from our initial 3D scene data. In the Imej library, we will be creating a fixed-function pipeline. This uses a fixed series of operations, each performed in order to create a render. We chose this method over a more flexible shader design due to performance implications. This may limit the use of our library as developers may only use the pre-defined functions provided. As we are designing this library for resource limited applications, high performance is essential. In using a well-defined process however, we are able to make assumptions about the execution of the pipeline. This will allow for stronger optimisations to be made at each stage.

3.2.3 Object transformations

To allow for objects to be manipulated in the 3D space, a set of transformation operations is required. These will include translation, rotation and applying a scale factor to an object. Such operations in 3D graphics are commonly applied through the use of a 4x4 homogeneous matrix[14]. To apply a transformation matrix to a three element vector, a fourth w term is added with a value of 1. The vector is then multiplied by the matrix, producing a new four element vector. To find the resulting three element vertex from this four element product, we use the last w term as a divisor for the other three terms. This division step will be essential later in this chapter for the creation of a projection system.

Within this structure, we can design a series of affine transformations. We use these affine transformations as a means to ensure parallelism and relative distances in a mesh structure are retained during the process.
A translation is created in this structure through use of the fourth column as shown in Figure 10. To apply a scale factor, the matrix seen in appendix Figure 12 Figure can be used. To support rotation using this system, we must decide the format used to call this operation. For the Imej library, we will take arguments in the form of Euler angles. A rotation operation is defined through each of its pitch, roll and yaw angles. Matrices which perform this rotation operation are included in appendix Figure 11.
The rotation and scale operations here can only define operations relative to the origin in world space. To apply a rotation about the center of an object, we must first perform a translation. We translate the mesh to the origin, apply the rotation, and subsequently apply a translation back to its original position. To combine these three operations into one operation, we can multiply the three matrices into one resulting structure.

In the header file which defines an objects transformation interface, a forward declaration will be used for its structure. Through using this technique instead of defining this using a matrix structure, we provide an avenue for further development of the system. Rather than our homogeneous matrix techniques, this could be replaced with a system oriented around a dual quaternion structure

3.2.4 Representing an object

As this library is centered around models in a 3D scene, we require a means of storing an object in memory. We settled on representing an object as a series of vertex and face data. Each face of the object is stored using an index to three points in the vertex list, which combine to a polygon. To perform transformations to an object, we must also store a transformation alongside each model.

As our library is designed to support multiple models concurrently, we define a datatype to store these. An ObjectList must implement functions to add and delete models from the scene, as well as provide a means of iterating through its contents. We use this iterator function rather than direct indexing to bring abstraction from the data structure. This will allow us to create and test two approaches. A fixed length array of pointers will be more performant, however, this requires a set limit on the object count. A linked list type structure can also be used, allowing for uncapped insertions but with a slight performance penalty.

One of the aims in this project is to allow mesh structures to be read from a machines file-system. To implement this, we have identified the Wavefront object (.obj) format as most suitable. This is a widely used, text based, open source standard, developed by Wavefront in the 1980s. Data stored in this format will be trivial for us to parse into an Object type[15]. Each line of an .obj file can store one of several data types. A vertex is identified with the character 'v' at the beginning of a line, followed by three numbers in scientific notation representing the x, y and z coordinates. A line storing face data begins with 'f', followed by three integers representing the index of the vertices which makeup this face. The Wavefront object format also has documentation for face normals as well as uv coordinates for texture mapping, however, these are beyond the current scope of the Imej project.

3.2.5 Viewmodel

For this library, we must create a means of moving the view through the 3D scene. A simple means of accomplishing this would be to apply a translation to every object in the scene when a movement is applied to the view. Whilst this technique would create the desired effect, it has several problems. In applying a translation permanently to the vertices which make up an objects mesh, its mesh can no longer be assumed as constant. Additionally, this technique may have performance implications as it requires many write operations to each vertex of every model. Instead, we can separate the view of the scene into a new object. This object acts as a camera in the scene, keeping track of its position and orientation in world space. Using this, we can change the location of our view through changing the points stored with the camera. To increment the view forward, we only adjust its position vector through adding a vector which represents the cameras facing direction. To create a image which depicts the view from the cameras position, we must create an operation which transforms the world space such that the camera is at the origin and oriented to be facing the positive z axis. This is known as a "world to camera" transformation. To create this, we must first calculate unit vectors representing right of the camera, up from the camera, and in front. Once these have been calculated, we place them in a rotation matrix and then apply a translation by the cameras position;

Figure 1: World to camera transform

$$\begin{pmatrix} x_{\text{right}} & y_{\text{right}} & z_{\text{right}} & 0 \\ x_{\text{up}} & y_{\text{up}} & z_{\text{up}} & 0 \\ x_{\text{forwards}} & y_{\text{forwards}} & z_{\text{forwards}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -x_{\text{pos}} \\ 0 & 1 & 0 & -y_{\text{pos}} \\ 0 & 0 & 1 & -z_{\text{pos}} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The camera object is also responsible for storing and handling data necessary for creating a projection of the 3D scene. To produce the 2D render of the 3D scene, a method is needed which can take 3D coordinates and return the pixel location on the framebuffer it maps to. Finding these integer pixel coordinates would require access to the dimensions of the display. In order to decouple the rendering pipeline from the rasterizer module, we instead produce Normalise Device Coordinates. These NDC coordinates define the frame as vertices within x(-1 to 1) y(-1 to 1) z(0 to 1). Once these are passed to the rasterizer, they can be interpolated from this range to integer coordinates within the display.

3.2.6 Orthogonal projection

To create an Orthogonal view, we will map points directly from the range within the cameras frame into the NDC range. To define area which is within the view of the camera, we use a view frustum. In defining the vertices which makeup this shape we can accurately identify if a vertex should be projected to the display. The frustum is created from six values. The cross section of the frustum relative to the camera is defined using a left, right, bottom and top values. For the z axis which represents distance from the camera, we use a near and far value. The near value defines the distance at which objects will begin to be drawn, with the far value being the cutoff point. One benefit of using this view frustum is we have also defined a clipping region. Vertices which lie outside of its area are known to be outside of view and can be discarded.

We can use the values which define a view frustum in our interpolation from camera space to NDC space through the following series of calculations;

$$X_{projected} = \frac{2 \times x}{right - left} - \frac{right + left}{right - left}$$

$$Y_{projected} = \frac{2 \times y}{top - base} - \frac{top + base}{top - base}$$

$$Z_{projected} = \frac{z}{far - near} - \frac{near}{far - near}$$

Using our homogeneous coordinate system, we can store these operations in a perspective matrix.

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-base} & 0 & -\frac{top+base}{top-base} \\ 0 & 0 & \frac{1}{far-near} & -\frac{near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3.2.7 Perspective projection

The projection created earlier creates a flat view of the scene. To produce a more accurate render, we must take perspective into account. As a model moves further from the camera, it should appear to shrink in the window. This can be accomplished through dividing each coordinate by its distance from the camera. In our homogeneous coordinate system, we use a fourth w term to describe a divisor for the resulting vector. Using this, we can construct a matrix which results in the w term being z, allowing for the depth division which creates a perspective view. A simple version of this projection would appear as the following;

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} x/z \\ y/z \\ 1 \\ w \end{pmatrix}$$

To map coordinates from the camera space to the NDC range, we again define a view frustum. Unlike our previous projection, this frustum does not have a constant cross section. To perform the shrinking effect, the cross section of the frustum is enlarged with distance from the camera. When points in the frustum are mapped to NDC points, points which are further from the camera are interpolated from a wider range. The cross section of the frustum is defined at the near plane. At this near plane, projected x coordinates must map directly from the range (left to right) to (-1 to 1). To apply perspective in the range defined by the near and far planes, we now multiply each point by $\frac{near}{zvalue}$.

In researching the construction of a perspective matrix, a Scratchapixel article was found which solves a similar problem[16]. The matrix created here, however, maps to a range of z coordinates from (-1 to 1), differing to our desired output of (0 to 1). For our circumstances, we derive a perspective matrix using the following calculations;

$$X_{projected} = \frac{x}{z} \times \frac{2 \times near}{right - left} - \frac{right + left}{right - left}$$
$$Y_{projected} = \frac{y}{z} \times \frac{2 \times near}{top - base} - \frac{top + base}{top - base}$$

To form a projection matrix in our homogeneous coordinate system for these equations, we create the following matrix;

$$\begin{pmatrix} \frac{2\times near}{right-left} & 0 & -\frac{right+left}{right-left} & 0 \\ 0 & \frac{2\times near}{top-base} & -\frac{top+base}{top-base} & 0 \\ 0 & 0 & A & B \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

To interpolate the z coordinate in this matrix, we can derive the A and B values. At the near plane, when z=near, the projected value must be 0. Similarly, at the far plane when z=far, the projected value must be 1. From this, we can form the equations:

$$A + \frac{B}{near} = 0$$
$$A + \frac{B}{far} = 1$$

Solving these gives us $A = \frac{far}{far-near}$ and $B = \frac{far\times near}{far-near}$, leading to the final matrix below;

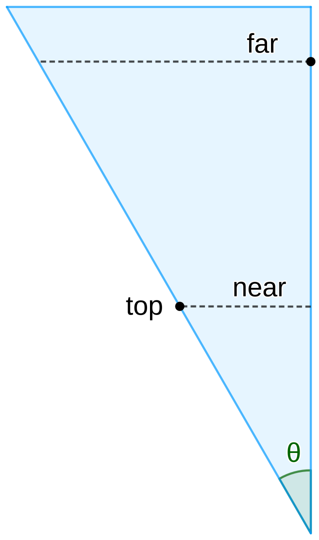
$$\begin{pmatrix} \frac{2\times near}{right-left} & 0 & -\frac{right+left}{right-left} & 0 \\ 0 & \frac{2\times near}{top-base} & -\frac{top+base}{top-base} & 0 \\ 0 & 0 & \frac{far}{far-near} & -\frac{far\times near}{far-near} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

For our library, we will use a view frustum which is always centered at (0, 0, 0) and symmetrical in the x and y axis. This means that our frustum will use a left value equal to -right, and top value equal to -base. We can use these properties to cancel terms in our perspective projection, simplifying to the following matrix;

$$\begin{pmatrix} \frac{near}{right} & 0 & 0 & 0 \\ 0 & \frac{near}{top} & 0 & 0 \\ 0 & 0 & \frac{far}{far-near} & -\frac{far\times near}{far-near} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

The values required to create this projection may be counter-intuitive for a user of our library. Instead, we could derive the planes which make up the view frustum from values given by the user. If a user provides a near and far clipping plane along with the desired field of view, we have sufficient information to construct the matrix. The top value can be found using trigonometric identities as seen in Figure 2. From this, we can find the horizontal components using the aspect ratio provided.

Figure 2: Deriving the 'top' value



$$top = \tan\frac{fov}{2} \times near$$

$$right = \tan\frac{fov}{2} \times near \times \frac{width}{height}$$

Using these, we come to the final perspective matrix;

$$\text{Let S} = \tan\frac{fov}{2}$$

$$\begin{pmatrix} \frac{1}{S} \times \frac{width}{height} & 0 & 0 & 0 \\ 0 & \frac{1}{S} & 0 & 0 \\ 0 & 0 & \frac{far}{far-near} & -\frac{near \times far}{far-near} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

3.2.8 Rasterizer

The rasterizer holds functionality to produce a bitmap image from our vertex coordinates. This will store the framebuffer memory and provide functions to interact with this. To draw a shape from the provided coordinate data, these must be rasterized into a set of integer pixel coordinates to be drawn. The draw functions provided will take values in the NDC range, i.e. x(-1 to 1), y(-1 to 1) z(0 to 1), and interpolate these into screen space coordinates, with a range of the screen dimensions.

To draw a line on the display, we must select an appropriate algorithm which produces the integer values of points on a line. One option would be to implement Bresenham's line algorithm. This requires some preprocessing to ensure a line has a positive gradient. A Differential Digital Analyser (DDA) algorithm could be used instead. This algorithm calculates a step value, and then periodically increments the x and y coordinate from the first vertex to the second.

To rasterize a triangle, there are several algorithms available to us. A common technique is to use scanline rendering. After sorting the triangles vertices by y value, we iterate over each y coordinate the triangle covers. The left and the right bounds at this y level are found from interpolating the two side edges. We then fill the line within these two bounds with solid colour.

Another technique which could be used is the algorithm described by Pineda in 1988 [17]. This algorithm first finds the bounding region around the triangle to be drawn. Next, each of the triangle's edges are translated into edge functions which can determine which side of the line a point lies. Subsequently, every pixel within the bounding region is iterated over. If the pixel lies on the correct side of all edge functions, it is drawn to the display. This algorithm is common in modern situations as it can be highly parallel, allowing for multiple threads to be used simultaneously. The Imej library is not designed to leverage multiple threads and as such, is unlikely to benefit from this approach.

To ensure faces which are obscured behind others are not drawn, a hidden surface removal mechanism is needed. This could come in the form of the Painter's algorithm. Faces are sorted according to their distance from the view, and then drawn in order from furthest to the nearest. This algorithm requires little memory to operate, however, it uses a large number of CPU cycles each frame to rasterize each and every polygon within the frame. This is on top of the sorting step which is required first.

Alternatively, a Z-buffer can be used. This is used to store the distance of each pixel from the view, allowing pixels which are closer than one already in the buffer to be drawn on top. Whilst this is an effective approach, it has a consequence of a large memory footprint, requiring a buffer with equal dimensions to the frame buffer. To alleviate this issue, the type of elements in the buffer can be replaced with one with a smaller memory size. Instead of storing each point as a 32 bit float, it could be replaced with a positive integer with a smaller number of bits.

3.2.9 Rendering pipeline contents

Earlier we provided a definition of a graphics pipeline. Here we will provide the stages which will be used for the pipeline in the Imej library.
When a new frame is requested, we must iterate over each object in the scene and attempt to render its mesh. To render an object, we iterate over each face defined in its mesh. From the vertices of a face, we follow the steps below to move from positions in object space to screen space coordinates which can be drawn to the buffer.

- 'object to world' : The transformation stored with an object is applied to a vertex, giving us the coordinate in world space.
- 'world to camera' : To migrate to camera space, we apply the 'world to camera' transformation. This migrates the coordinate space such that the viewmodel is positioned at the origin.
- clipping : Here we can use the bounds of the view frustum to identify if a vertex should be discarded and continue the process with the next face.
- projection : We next apply the projection matrix to our vertex.
- 'camera to screen' : Points which were in the NDC range are interpolated to fit the dimensions of the framebuffer.
- rasterize : The resulting polygon is drawn on the framebuffer.

3.2.10 Display

The display module implements a means of presenting a frame generated by this library. If the medium offers support for user interaction, such events can be captured here and forwarded to the input handler. This includes the main continuous loop. Its structure includes checking for input, calling the input models tick function to process the view's momentum, requesting a frame to be rendered and then presenting the frame.

4 Implementation

During the implementation stage, Beej's guide to C programming was a valuable resource[18]. This resource contains a comprehensive description of the C language, covering a majority of its features. This extended our knowledge of the language, which enabled us to create a cleaner codebase. In particular, its documentation for structs and unions was essential for the creation of Imej.
For example, we use a union to define a 32 bit colour.

```c
typedef union{
  uint32_t colour;
  struct{
    uint8_t b;
    uint8_t g;
    uint8_t r;
    uint8_t a;
  };
}colType;
```

This allowed us to set each each of the r, g and b channels directly rather than using a mask over the 32 bit representation.
Similarly, we created a union for a 16 bit colour using the 545 format. Through using bitfields, we can accurately index the bits used for each channel within the 16 bit colour.

```c
typedef union{
  uint16_t colour;
  struct{
    uint16_t b: 5;
    uint16_t g: 6;
    uint16_t r: 5;
  };
}colType;
```

4.1 Vectors and matrices

For our library to function we needed to implement handling for common vector and matrix operations. Initially, matrix multiplication was implemented through a series of nested loops with the following logic;

Using the online tool Godbolt, we can view the assembly generated by GCC for a section of C code, included as Figure 1 in the appendix. This first

```
float resulting matrix[4][4];
for $i = 0$ to 3 do
    for $j = 0$ to 3 do
        for $k = 0$ to 3 do
            resulting[$i$][$j$]$+=$ matirxA[$i$][$k$] $\times$ matrixB[$k$][$j$];
        end for
    end for
end for
```

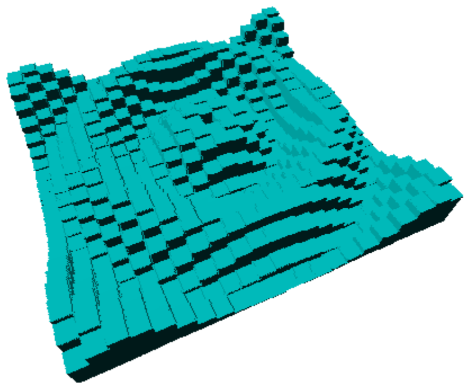
requires each element of the resulting matrix to be set to 0. After this, each nested loop requires its counter to be initialised, and a jump to the loops contents. Within each loop block, the counter is incremented and a conditional jump instruction used as an exit condition. This control flow requires an unnecessary number of CPU instructions, a problem for us given that this represents a function with a high call volume in our library. To improve this algorithm, we can utilise loop unrolling. If we replace this algorithm with one which instead sets each of the 16 elements in the output matrix directly, we can drastically improve its execution time. To test this, we created a script which performs 10 million multiplication operations. With the previous algorithm, this took 2.8 seconds to execute. The new version completes these operations in only 0.3 seconds, giving a 10-fold performance improvement.

4.2 Models

To store a model, we must create a datatype which can store the mesh and colour data for an object in memory. Each model has a mesh structure, containing lists for its vertices and face data. The vertices used for this mesh initially used our vec3 type, which stores coordinates using 32 bit floats. For models with a complex mesh structure, this approach had a large memory footprint. To reduce this issue, we created a new type for the vertices in a mesh. This instead uses 16 bit integers for the vertices of a mesh. To achieve this, an object is scaled such that its mesh takes the full range provided by the 16 bit integer. To find the position of the object in the scene, these points are divided by the maximum value, leaving us with an object with coordinates ranging from -1 to 1. In order to place this object in the scene, we now store the transformations applied to an object alongside it. When a new transformation is applied, it is instead multiplied to the objects existing transformation. To find the position of an object in the scene, we apply this transformation to each vertex as it is processed by the rendering pipeline. This change effectively halves the memory required to store a mesh structure. Using the full range available with a 16 bit integer, we can store vertices of a mesh across a range of 65535 in each dimension. This change brings another benefit. As we are using the entire range offered by a 16 bit integer for an objects vertices, we are able to find the minimum cube in world space which completely envelops the object. We will take advantage of this property to perform efficient culling in the projection process detailed later.

To allow for the creation of animated models, each model has an additional callback function. If this callback function is set, it is called with each frame. This function allows for the creation of animated models in a scene. One use case would be to simply apply a transformation to a model between frames, such as creating a continuously rotating object, however, its use can extend further. Our new approach for storing a models mesh structure allows us to make changes in model space. Rather than updating the vertices of a mesh during execution, we instead store its transformation and use this to find its world space coordinates. This means a developer can modify an objects mesh structure during runtime without considering any transformations which have been applied to it. Using this property, we can create software which changes a models mesh structure in real time. One example of this is a graphing utility. A mesh can be created which projects a 2D array of data as a 3D bar chart. Between each frame, this mesh can be updated to show the most recent data available. Figure 3 shows this utility being used to plot a sin wave which oscillates over time. This utility has applications for real world embedded systems. Data collected such as temperature or resource usage can be presented to a user intuitively through a graphical representation.

Figure 3:



Model container

For our library to support multiple 3D meshes in one scene, we require a means of storing several objects in memory. To allow for this to be implemented using a variety of techniques, it has intentionally been designed with a flexible signature. In the model container header file, we have used a forward declaration for the container, allowing its structure to be defined in the implementation. Additionally, the functions which are used to interact with this structure are designed to create an abstraction between the interface and the container. To iterate over a container, we have designed an iterator function to behave similarly to Python's generator object. To iterate over the models in a scene, the function is first called with the container pointer. Subsequent calls with a NULL argument will return each subsequent object available, with a NULL return value signifying the list has terminated.
For our implementation, we used two approaches, a static array and a linked list structure. For a fixed length array, we must define the maximum number of objects at compile time. Insertion to this structure has O(1) time complexity. A variable is used to point to the next available index. An objects reference pointer is stored here and the counter incremented. Similarly, iteration over this structure is simple, with each pointer being sequential in memory. A problem with this solution however is it requires a fixed length array. Once the structure is filled, no new models can be added to the scene, meaning we must decide its requirements at compile time.
An alternative to this is to use a linked list structure. Here each element in the container stores a reference to the next node in the chain. Using this, we can create a structure which can grow to support any number of models in the scene. This approach brings more complexity to our iteration. Rather than simply indexing the next element in the array, we must instead follow a chain of pointers through memory.
We can combine these two approaches to create a system which features both fast iteration and an unlimited size. To create this, we use a structure similar to the linked list, but with each node instead storing a fixed length array of elements. Using this, we are able to quickly iterate over a small set of objects in a node, but have the flexibility of a dynamic data structure if a use case requires a large number of models. We can compare the performance of these solutions through running a benchmark. To test these approaches, we will benchmark their performance with the insertion on 4,000 elements followed by iterating over the contents 4,000,000 times.

Table 1: Benchmarking container structures

Type	Insertion	Iteration
array	0.034ms	71.09s
linked list	0.23ms	75.51s
combined solution	0.175ms	69.62s

Model optimisation

During the implementation, another technique was identified to reduce memory usage in certain situations. Creating multiple copies of an object in a scene may prove to be a common task. In our current design, this would mean the vertex and face data for a model would be stored twice in memory, despite these having identical contents. For these situations, we instead add a clone function to our interface. To clone an object, we create a new model but set its vertices and face variables to point to the object which is being cloned. To facilitate this operation whilst ensuring memory is managed properly, we must add a shared reference counter variable to each mesh. This is initially set to 1, and then incremented with each clone an object undergoes. To free an object from memory, we perform a check against this shared counter. If it equals 1, it is safe to deallocate the mesh alongside an object. If not, we instead decrement its value and leave the mesh data in memory for the cloned objects to reference. Using this clone function, we can test the memory implications for a scene containing 200 unit cubes.

Table 2: Memory usage of a cloned object

Method	Memory usage (Kib)
without cloning	105.5
using cloning	30.1

4.3 Camera

This module is responsible for handling everything related to the view of the scene. This includes the position and orientation of the camera, but also includes a projection matrix and the vertex makeup of its view frustum.

Orientation

To store the location and orientation of a camera in the scene, we use a position vector and store pitch and yaw angles. During the rendering pipeline, a transformation from world to camera space is required. To construct the matrix seen in Figure 1, we first calculate vectors which describe the orientation of the camera. A forwards vector can be calculated from the angles stored with the camera.

$$x_{\text{Facing}} = \cos yaw * \cos pitch$$

$$y_{\text{Facing}} = \sin pitch$$

$$z_{\text{Facing}} = \sin yaw * \cos pitch$$

The right vector is found through applying a cross product to an up vector and the facing vector. Once this matrix has been calculated, we store it with the camera. When the world to camera transform is required, if no change is made to the camera, we can return the previously calculated transformation.

Projection

In the design stage, we derived a projection matrix which maps world space coordinates into NDC space. When monitoring the performance of our library, the subsequent conversion from NDC space to world space was identified as a bottleneck. Every point generated by the projection matrix is passed to the rasterizer, for it to then be interpolated into a screen space coordinate. This represented an intensive operation, which was performed for every vertex in view. We can improve this system through adding the ability for the rasterizer to take screen space coordinates directly. The interpolation to screen space can be combined with our projection matrix, allowing the two operations to be performed as one. For this, we must make some changes to our projection matrix. This interpolated includes flipping the y coordinate to align with the framebuffer, which uses (0, 0) as the top left coordinate.

$$X_{projected} = \frac{x \times width}{right - left} - \frac{width \times left}{right - left}$$

$$Y_{projected} = \frac{y \times height}{top - base} - \frac{width \times base}{top - base}$$

$$Z_{projected} = \frac{z \times zMax}{far - near} - \frac{near \times zMax}{far - near}$$

This forms the orthogonal matrix;

$$\begin{pmatrix} \frac{width}{right-left} & 0 & 0 & -\frac{width \times left}{right-left} \\ 0 & \frac{height}{top-base} & 0 & -\frac{height \times base}{top-base} \\ 0 & 0 & \frac{zMax}{far-near} & -\frac{near \times zMax}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Similarly, our perspective matrix can be adapted to provide values in the screen space.

$$\text{Let S} = \tan \frac{fov}{2} \begin{pmatrix} \frac{1}{S} \times \frac{height}{2} & 0 & \frac{width}{2} & 0 \\ 0 & -\frac{1}{S} \times height2 & \frac{height}{2} & 0 \\ 0 & 0 & \frac{far \times zMax}{far-near} & -zMax \times \frac{near \times far}{far-near} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

4.4 Rasterizer

The rasterizer is used to calculate the discrete integer coordinates covered by a shape on the frame. This required a function to render a line between two vertices as well as a function for a triangle made of three vertices. The line drawing algorithm was implemented using both Bresenham's line algorithm and a DDA style algorithm.

4.4.1 Z buffer

To prevent faces which are obscured behind ones closer in the scene we use a z-buffer. In our perspective matrix, the z coordinate is projected non linearly as shown in Figure 4. This brings a benefit when using the z-buffer. Coordinates which lie closer to the camera map to a wider range of z values. This results in our occlusion checking being more accurate for models which are the most important in the scene.

4.5 Rendering pipeline

Our rendering pipeline process a vertex through several stages. We move through each of these stages by applying a matrix transformation to the vertex. An initial version of this has the following form; This process requires several matrix multiplication operations to be performed. For each vertex of

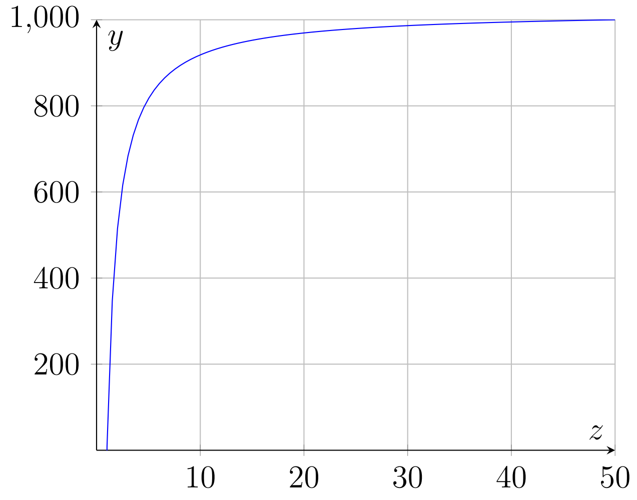


Figure 4: z coordinate projection

Algorithm 1 Pseudocode for processing each object in the scene

```
for all object in the scene do
    for all face of an object do
        apply object to world transformation
        apply world to camera transformation
        clip the vertices to the view frustum
        apply a perspective projection
        draw the resulting face on the framebuffer
    end for
end for
```

each face, we must apply three different transformation matrices. Whilst this represents many multiplication operations, a major problem is the division performed after each, a costly operator. We can improve on this process through combining these matrices before we begin the iteration. Using this approach, we must only perform one multiplication within the loop structure. To test these two approaches we created a benchmarking program. This constructs a scene with 1000 models equally distributed around the view. For the model, we use a Utah Teapot mesh, which features 3644 vertices and 6320 faces. The frames produced during this benchmark are drawn to a 500x500 pixel framebuffer. To test these algorithms, we use them to produce 1000 frames and record the time taken.

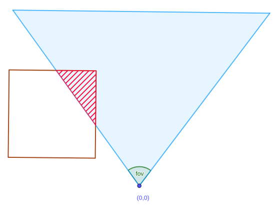
Table 3: Radius calculation

Pipeline	execution time
seperate operations	88.28s
combined transformation	58.99s

Clipping a face

To ensure the points we send to the rasterizer are within the bounds of the frame, we perform a clipping operation. The Sutherland-Hodgman algorithm allows us to perform this clipping whilst also creating a new polygon which lies within the frustum[19]. To perform this on a face, we must iterate over each pane of the view frustum. For each edge of a face, we check which side of the frustum it lies. If the edge is entirely infront of the plane, we do not need to perform clipping. If the edge is partially behind the plane, we discard the offending vertex and replace it with one at the intersection of the edge and the plane. Using this, we can create new polyshapes which represent the area of a face within the view frustum as shown in figure 5.

Figure 5: Clipping a shape against a view frustum in 2 Dimensions



During the implementation, we performed an analysis of the libraries execution using Valgrind Callrind. Using this, we can identify the number of times a function is called during execution of a function. In our testing, this clipping function was one of the most used, representing 14% of all function calls. From this, we can identify the clipping process as a potential bottleneck for our library, meaning any optimisations are critical. One improvement we can make is to instead apply this process after the projection has been applied. Currently we must define the clipping planes in three dimensions using its normal. After the projection has been applied, the planes of our view frustum become paralell. Using this, we can check the vertices of a face against the range of the projected view frustum. With this, we can continue without performing the clipping operation if it is not required. Additionally, the process of performing clipping can be improved. Rather than computing using the normal of a plane, we instead use the range of values in each dimension set by the projected frustum.

Culling objects

To improve the speed of our rendering process we must identify faces which should not be drawn as early as possible and skip their processing. One technique we use is known as Backface culling. Once a face enters the world space, we can check if its face is oriented towards the camera. To do this, we find the dot product of the face normal and its position relative to the camera. If this is greater than 0, we know the face is occluded and it can be discarded.
Due to the optimisation we made to our rendering pipeline earlier, we skip past the world space value for each vertex. We must now perform this culling in the camera space. This allows for a simpler solution. If the z element of a face normal is negative, it is oriented away from the camera and can be discarded.

Another method to reduce unnecessary processing is to perform an early check of a whole mesh against our view frustum. Earlier we made the decision to limit the vertices of a mesh structure to the bounds (-1,1), we have created a bounding region for each shape. Using the objects transformation matrix, we can find the points which describe its bounding box in world space. These points can then be used to verify if the object is entirely outside of the view frustum. For each plane of the view frustum, we take the dot product its normal and the closest bounding point. If this value is greater than 0, then we can conclude no part of the mesh structure is within view and the object can be ignored. The performance gains from this approach vary depending on the contents of the scene. This has the largest impact for scenes with high poly models which are likely to exit the frame. In allowing an object to be ignored before its mesh is considered, we skip the process of iterating over each of its faces. To showcase the difference this approach can make, we will construct a benchmark for an ideal situation. For this, we create a scene containing 216 copies of the Utah teapot model, evenly distributed around the camera. The table below shows the time taken to produce 1000 frames to a 500x500 pixel buffer with and without this culling technique.

Table 4: Benchmarking object culling

Test	Time for 1000 frames	Average FPS
without culling	248s	4.03
using culling	21.0s	47.6

4.5.1 GCC

Earlier we found performance gains through manually unrolling loop structures. The GCC compiler we used for the development of Imej has the ability to perform similar optimisations as well as many more. Through setting the optimisation level using the -o3 flag, we can instruct the compiler to perform all optimisations available. This comes at the expense of compilation time and can lead to a potentially larger executable[20]. With these flags enabled, the performance of the Imej library is more than doubled.

4.6 Display

To display the rendered image to a user, an implementation of the "window.h" interface must be implemented. This provides a means of handling memory related to a window. Functions are required to open, close and push a frame to a window, as well as a function which begins an infinite rendering loop. If a medium supports user interaction, such events must be captured here. To process these events, a separate module for input handling is used to update the view. This interface is a key part in the portability of this library. To offer support for a new visualisation medium, a custom implementation must be created for only this header, allowing other sections of code to be reused. To demonstrate this library being used in a range of situations, we have provided several implementations for the display header. Each of these allows for frames produced by the library to be presented in a unique way.

X11 GUI

To create a traditional GUI window on machines running the Linux operating system, we have created a driver for the X windowing system. This interacts with the X server to open a window and push the framebuffer. Events provided by Xlib can be captured and processed if user interaction is enabled. The events supported include mouse and keyboard operations, allowing for convenient user interaction with a simulation. This simulation uses full 32 bit colour to create a high resolution image on supported machines.

Terminal emulator

To offer visualisations for machines without a bitmap display, we offer a means of producing an image in a terminal. We create this image through use of ANSI escape characters. Terminal emulators which support this technology allow for the colour of each character to be set using our 32 bit value. To print a pixel, we can use unicode block characters to simulate a solid pixel in the window. We can find the dimensions of a terminal on supported machines through a syscall offered by ioctl. If the dimensions of a terminal are larger than the framebuffer, we can map the image directly to the window. If this is not the case, we can instead sample a region of pixels from the buffer, averaging their colour to produce a character to print.

PPM image

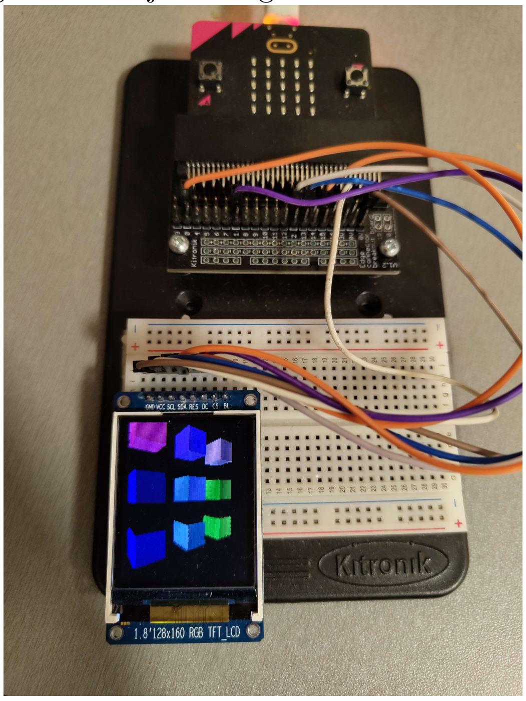
To provide a means of saving produced frames to non volatile storage, we have created a driver which uses the PPM image format. The Portable Pixmat format offers a simple means of writing bitmap images to a file. To capture a video output, we use the rendering loop function to create a series of frames. Each frame is saved to a frame folder, using an incremental counter for its filename. We have included a shell script with the Imej library which can be used to create a .gif file from this series of frames. This uses the Convert command line utility offered by ImageMagick which allows these frames to be combined.

BBC Microbit

As part of this project, we had the intention of running a 3D graphics program on the Microbit. There were two main options available to us to display a produced frame. The first would be to extend the terminal driver we created earlier. Frames produced on the board could be sent through a serial connection to a connected machine. This could be created from scratch through a bitbanging techniquie or instead through using the provided uBit object for serial communication.
Instead of this, we chose to connect a LCD display to the board. Using this, we can display images generated on the Microbit without needing a machine to be connected. For this, we used an Adafruit TFT LCD screen with 128 x 160 pixels. Commands are sent to this display through an SPI connection, using a modified version of the ST77255 header provided by Adafruit. To compile a project for use on the Microbit, we use the build process provided with Codal. The header files and an implementation for each are moved to

its target directory, along with the LCD display headers. Rather than 32 bit colour, this display uses a 565 format for RGB colour, with each channel restricted to 5, 6 and 5 bits respectively.
With the creation of this driver, we have included a means of interacting with a visualisation. If user input is requested, the Microbit will listen for keyboard events provided over a serial connection. This allows a connected machine to act as a controller for an application.

Figure 6: Imej running on the BBC Microbit



5 System in operation

Alongside the source code provided for this project, we have created sever tools to assist with development.

5.1 Compilation

A project using Imej can either be compiled manually using the implementations provided, or as a static library using the provided makefile. This makefile has several targets. 'make imej' will compile only the Imej library. This produces a archive file for the library, which can be later used through linking with an application. Also provided is 'make link FILENAME=filename'. This command takes a path to a project which uses Imej. In addition to the Imej library, the provided program will also be compiled and linked by the makefile.
Several arguments can also be used with this makefile to customise the behaviour of Imej. The default projection used is a perspective view, however setting IMEJ_PROJECTION to orthogonal will instead use a orthogonal view. IMEJ_DISPLAY can be used to select from one of the provided display mechanisms. Options include 'ppm', 'x11' and 'term'. User interaction is enabled by default, however it can be removed through setting IMEJ_INPUT to false. This library supports both 32 bit and 16 bit colour schemes. In most situations, the type used will be defined by the display medium, however these can be set manually through setting IMEJ_COLOUR to IMEJ_32_BIT_COLOUR or IMEJ_16_BIT_COLOUR.

5.1.1 Shell script utilities

For quick demonstration of Imej, we have also provided the 'exampleUsage' shell script. When this is executed, a user is able to select from a list of example models to download. Once a model is acquired, a series of demonstration programs can be selected here, showcasing the libraries function. For the PPM demonstration, we have included additional tools. Choosing this option will launch a program and generate a series of frames, each stored sequentially to a folder. Once this has completed, these are combined into a gif file using the imagemagick shell utility.

5.1.2 Targeting the Microbit

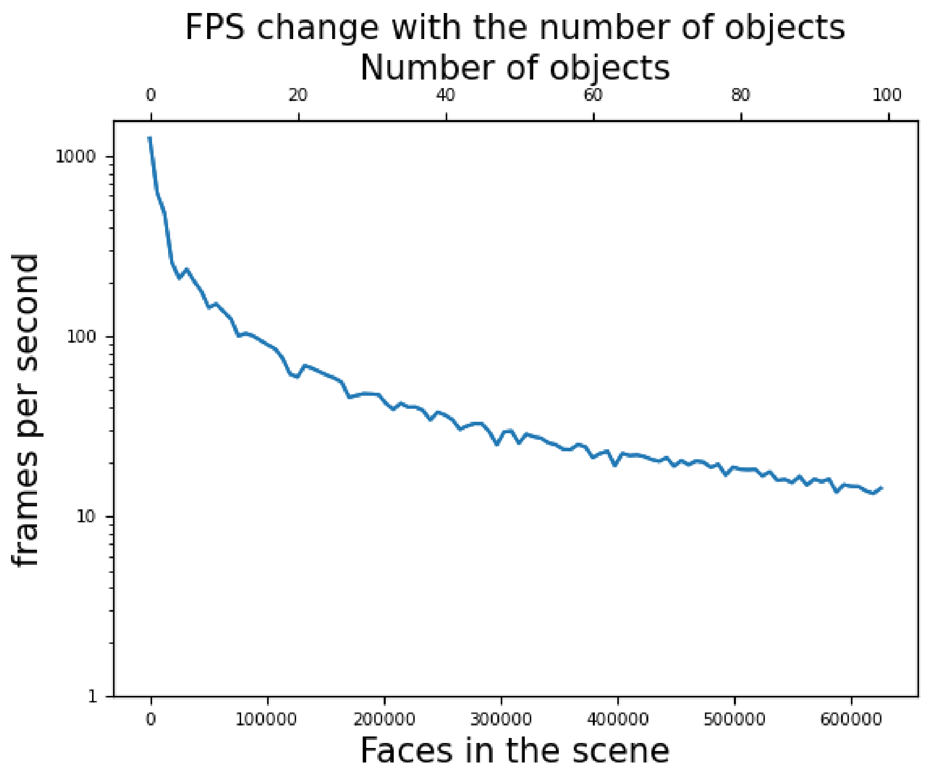
To run the Imej library on the BBC microbit with a TFT LCD we require several dependencies. To interface with the display over SPI, the Codal runtime is required. Additionally, a version of the Adafruit TFT driver, ported for the Microbit by Joe Finney is required. Once these files have been downloaded, the library can be used through including the imej.h header provided and compiling with the necessary source files in the same directory.

6 Testing

Our first set of tests will be performed on a Thinkpad T420. This represents a comparatively powerful machine, with 3.4GHz clock speed and 16Gb of memory available.

6.1 Framerate

Our first benchmark will be to measure the effect models in a scene have on the framerate achieved. To measure this, we construct a benchmark which produces a 500x500 pixel framebuffer. Our test will incrementally add models to this scene, and then measure the time needed for a number of frames. For this, we will again use the Utah Teapot model. To remove the impact object culling will have on the rendering process, the models will be added to a grid which is within the view frustum. The results of this benchmark can be seen in Figure 6.1



For our previous test, no changes were made to a scene between frames. To test the performance of an animated scene, we will construct a dynamic simulation. For this, we will provide an update function to our mesh which will be called between each frame. For this, we created a graphing utility. To use this, a 2d array must be provided containing 16 bit integer values. Returned is a mesh structure for a bar chart of the data. As a mesh is defined in object space, we can update its structure at any point without being affected by its location in the scene. To create a benchmark using this, we will plot a 3D sin wave, updating its structure each frame. A mesh created using this can be seen in Figure 7

Figure 7:

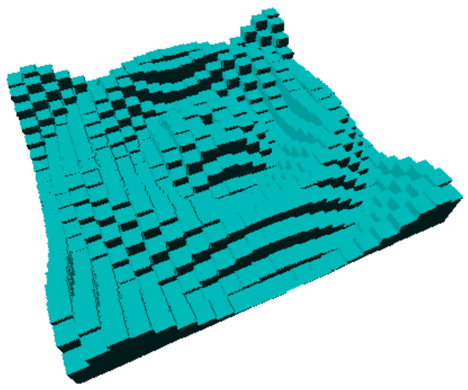
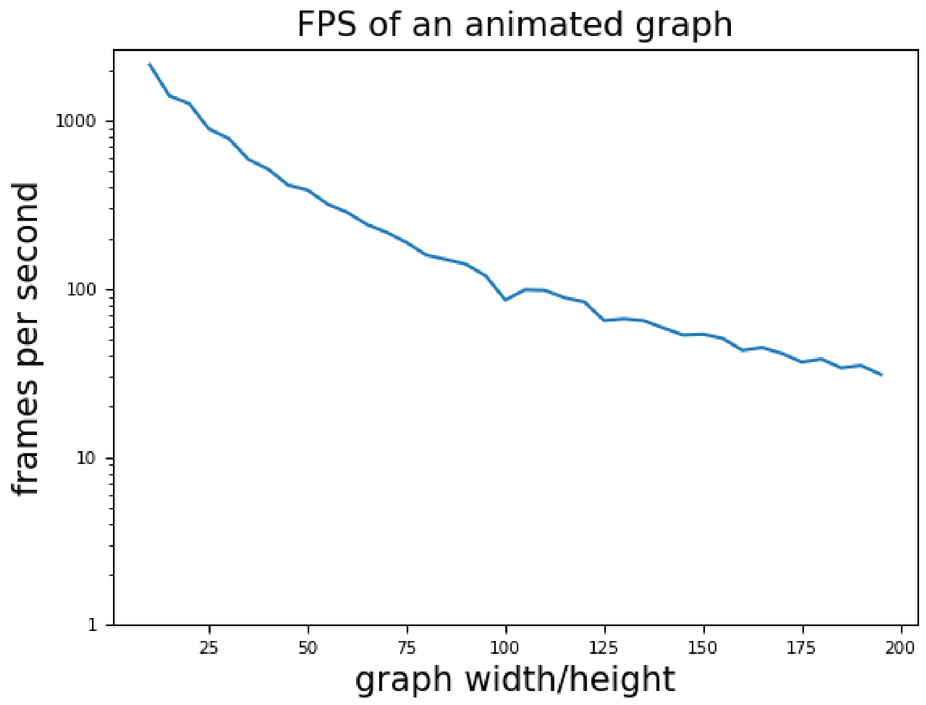


Figure 8 shows the results of plotting such a graph with a range of dimensions.

Another variable we have not included so far is camera movement. In each of these tests, the camera has remained stationary in the scene, meaning the world to camera transformation is never changed. To test the impact of a moving camera in a scene, we will run the same benchmark twice. As the camera transformation is requested once per object, for it have the highest effect on performance we will use a scene containing on cube model. To gauge the impact of a moving camera, we will first run the simulation with a stationary view, recording the average fps. For the second test, we will translate and rotate the view slightly between frames, ensuring the camera to world transform will be recalculated. The results in Figure 9 show that changes to the view of a scene have a negligible impact on performance.

Next we will measure the performance of our library when running on the Microbit. For our first test, we will use a scene with a single cube which continuously rotates. Measuring over 10000 frames, we recorded an average of

Figure 8:



40.7 fps. To determine the impact of the SPI communication on performance we can perform this test again. This time, each frame is produced to the buffer but is not sent to the LCD display. With this change, the fps increased to 76.2.

For a more intensive test, we will again use the animated graph detailed earlier. For this, we create a sin wave animation, using a 10x10 grid of cells. When animating this on the LCD display, we measured an average of 17fps across 1000 frames.

6.2 Memory usage

The memory used by this library largely depends on the dimensions of a produced render. The frame and depth buffer account for most of the memory used during runtime. The memory used by these can be calculated from the

Figure 9: The impact of a dynamic camera on performance

Test	average FPS
Stationary camera	2069.3 fps
Dynamic camera	2059.2 fps

dimensions of the display and the colour depth which is used.

$$\text{Frame buffer} = \text{Colour depth} \times \text{width} \times \text{height}$$

$$\text{Depth buffer} = 16 \text{ bits} \times \text{width} \times \text{height}$$

For example, a program which uses 32 bit colour values and has dimensions 1366 x 768 would use 4.19Mb for the frame buffer and 2.0Mb for the depth buffer. To verify this with an actual program, we can record memory usage with Valgrind Massif.

```
n1: 4196352 0x109B90: createRasterizer (raster.c:38)
 n1: 4196352 0x10A30C: createScene (scene.c:44)
  n0: 4196352 0x10916C: main (debug.c:32)
n1: 2098176 0x109B78: createRasterizer (raster.c:34)
 n1: 2098176 0x10A30C: createScene (scene.c:44)
  n0: 2098176 0x10916C: main (debug.c:32)
n0: 3652 in 7 places, all below massif's threshold (1.00%)
```

Objects in the scene are the next memory concern. To reduce the footprint of a mesh, their vertices are stored with a 16 bit integer instead of a 32 bit float. The size required for an object can again be predicted. Each object instance uses 32 bytes, plus 64 for its transformation if one is used. A mesh structure requires a base of 32 bytes, plus 6 per vertex and 6 per face. The memory required for an object can be found using Memory = $32 + 32 + (6 \times \text{vertices}) + (6 \times \text{faces})$. For example, a unit cube which has 8 vertices and 12 faces would require 184 bytes of memory. A higher poly model such as our Utah Teapot which has 3644 vertices and 6320 faces requires 59.85Kb.
We can consider the implications of this when using Imej on the Microbit. Available to us is a total of 128KB of memory. Our LCD Display has dimensions 128 x 160, meaning the frame and depth buffers require 81.9KB. If we ignore the memory used by other elements in the library, this leaves us with 39KB to store models. Within this, we can theoretically store 153 unit cubes. In reality, we can reach a maximum of 87 models before the heap is filled. This number can be increased if we instead use the clone function,

sharing the mesh for each cube. Using this, we can reach 240 cube models in a scene before the heap is filled.

7 Conclusion

7.1 Review of aims

Overall, we have met the aims initially set out for this project. In creating the Imej library, we have met our main objective of creating a 3D graphics engine from the ground up. To achieve this, we had several smaller objectives to meet. The ability to modify objects in the scene has been created through three operations. The scale and translation operations were created successfully and fully meet the requirements. To meet the aim of allowing rotation operations, we created a system which uses Euler angles. In theory, this allows for full control as any rotation operation can be represented through the three pitch, roll and yaw angles which are required. We could have provided a more flexible interface however. Rotation operations could have instead been described using a rotation angle about a vector, allowing for rotations to be created around axis other than the orthogonal axes.

Another key aim for the Imej project was for its design to allow for easy extension. To meet this objective, we integrated several techniques into its design. Each distinct set of functionality has been split into a separate module. The interfaces of these modules were created around a high degree of abstraction. Functionality such as the transformation system use forward declarations and abstract interfaces to allow for unique implementations. For some of our interface however we could have provided more flexibility. If we included a more polymorphic design for our structures, developers would have the ability to create multiple implementations for the same process. A possible extension for this project would be support for texture mapping over a face. If we had opted to create our face type with a vtable to indicate its draw method, a developer could extend the face type and include a specific draw function which applies the texture provided.
Another decision which could have improved the flexibility of our project is to use a publisher subscriber pattern for events. With our current design, for each frame the input handler must be ticked to process the inertia of the camera. Additionally, to support the frame callback for an object, we must check if an object has a callback function and if so trigger it when we iterate through the scene. If we had instead used a event based model, these operations could have been controlled through subscribing to a frame event. Rather than storing a callback function with each object, an object which needs this feature could instead just subscribe to the frame event, potentially reducing memory usage. Additionally, this would mean the input tick would not have to occur in the main loop for our program, decoupling these two

modules.

Another aim for the Imej project was to demonstrate its flexibility through offering support for several display mediums. This aim has been thoroughly met, with support for a range of situations offered in this submission. A driver for the X windows system allows for a traditional GUI window to be used. The PPM implementation we created allows for frames to be captured and exported to disk. As well as these, we have demonstrated the library being used with an LCD display connected to the BBC Microbit device.

7.2 Further development

For further development of the Imej project, there exists a range of possible additions. The current state of the library has several avenues for improvement. As mentioned in the review of aims, its portability could be improved through the use of vtables to allow for polymorphism. Additionally, including an publisher subscriber event model would further decouple some of its modules. As part of the rendering process, we currently apply the projection to each vertex of a face, and then find its normal to perform backface culling. If we instead compute the transposed inverse of the matrix, we could instead compute the normal of a face in object space and apply the transformation to this. This technique could potentially improve performance as backface culling can occur before all 3 vertices of a face are projected.

Another potential improvement is a means of indicating when a change has occurred in the frame. Rather than recomputing the framebuffer continuously, this would allow us to retain the previous frame when no changes have occurred to the scene.
In terms of additions to the library, there are a range of possibilities. Adding support for texture mapping would allow for more detailed models to be created, although the memory needed for each texture would have to be carefully considered. More work could be completed on the transformations the library uses. Operations completed in world space have the potential of being replace with quaternion representations, potentially using less memory for each object.

7.3 Learning outcomes

The development of the Imej library had given me a much broader understanding of the C programming language. Its development was a much larger task than initially expected, with any minor bug leading to a complete failure of the rendering pipeline. Despite this, I found it a thoroughly enjoyable experience. Going from an initial program which struggled to reach 1fps to the final optimised offering was truly rewarding. Its development required me to learn new tools and techniques. I have learnt how to use profiling tools to identify locations which would benefit from better optimisation. To identify bugs before they became an issue I learnt the process of unit testing C code with Check.

The research involved in creating the Imej library has been a valuable endeavour. At the beginning of this project I had a limited understanding of the concepts used in the rendering process. The process of completing my own 3D graphics engine has taught me the function and use of matrices and their applications in computer graphics.

I have used LaTeXfor typesetting documents in the past, but never on a project of this scale. Producing this document taught me more about its build process and the ecosystem which surrounds it.

7.4 Final remarks

To conclude, this project was successful in creating a 3D graphical system. Its design is centered around low resource use, with simple simulations being able to run in real time on the BBC Microbit board. I have enjoyed every moment working on this library, and will likely continue development into the future.

References

[1] William J. Dally, Stephen W. Keckler, and David B. Kirk. Evolution of the graphics processing unit (gpu). *IEEE Micro*, 41(6):42–51, 2021.

[2] Ping-an He, Dan Li, Yanping Zhang, Xin Wang, and Yuhua Yao. A 3d graphical representation of protein sequences based on the gray code. *Journal of theoretical biology*, 304:81–87, 2012.

[3] Samuel Axon . "forging new paths for filmmakers on ẗhe mandalorian"̈. Accessed March 2024. `https://www.unrealengine.com/en-US/blog/forging-new-paths-for-filmmakers-on-the-mandalorian`.

[4] John F. Hughes et al. *Computer Graphics: Principles and Practice.* Addison-Wesley Professional, 3rd edition, 2013.

[5] L. Snyder and National Science Foundation (U.S.). *Proceedings of the Conference on Experimental Research in Computer Systems.* National Science Foundation, 1997.

[6] Nick Evanson. 25 years later: A brief analysis of gpu processing efficiency. Accessed March 2024. `https://www.techspot.com/article/2008-gpu-efficiency-historical-analysis/`.

[7] H. Fink, T. Weber, and M. Wimmer. Teaching a modern graphics pipeline using a shader-based software renderer. *Computers and Graphics*, 37(1):12–20, 2013.

[8] Pawel Lapinski. *Vulkan cookbook: work through recipes to unlock the full potential of the next generation graphics API-Vulkan.* PACKT Publishing, 1st ed. edition, 2017.

[9] History of opengl. Accessed March 2024 `https://www.khronos.org/opengl/wiki/History_of_OpenGL`.

[10] Nvidia. Vulkan. Accessed March 2024 `https://developer.nvidia.com/vulkan`.

[11] Jacob Roach. What is directx, and why is it important for pc games? Accessed March 2024 `https://www.digitaltrends.com/computing/what-is-directx/`.

[12] The standard for embedded accelerated 3d graphics. Accessed March 2024, `https://www.khronos.org/opengles/`.

[13] Valgrind. `https://valgrind.org/`. Accessed: March 2024.

[14] Feng Lu and Ziqiang Chen. A general homogeneous matrix formulation to 3d rotation geometric transformations. *arXiv.org*, 2014.

[15] Paul Bourke. Object files. Accessed March 2024 `https://paulbourke.net/dataformats/obj/`.

[16] The perspective and orthographic projection matrix. Accessed March 2024, `https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/opengl-perspective-projection-matrix.html`.

[17] Juan Pineda. A parallel algorithm for polygon rasterization. *Computer graphics (New York, N.Y.)*, 22(4):17–20, 1988.

[18] Brian Hall. Beej's guide to c programming. `https://beej.us/guide/bgc`.

[19] Thomas Revesz. Clipping polygons with sutherland-hodgman's algorithm. *The C users journal*, 11(8):23, 1993.

[20] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 2008 CGO - Sixth International Symposium on Code Generation and Optimization*, pages 165–174. ACM, 2008.

A Appendix

A.0.1 Project proposal

SCC300
Final year project proposal

Dexter Latcham
38380447

23-oct-2023

Contents

1 Abstract

The project outlined in this paper aims to produce a lightweight and efficient 3D graphics library tailored for embedded systems. This library will facilitate the loading and manipulation of objects in a 3D space, rasterizing them to a 2D plane. By default, this will be displayed on a GUI window however using alternate means of visualisation will be possible.

The completed product will be designed with extendability and portability in mind. Through code separation and clarity, it should be trivial for a developer to adapt the end product to suit their needs.

2 Introduction

In the current digital age, computer generated graphics play a dominant role in a wide range of industries. In the UK, the video gaming industry alone has a market cap of £7.16bn [1], however the influence of this technology extends far beyond, from architecture and product design to DNA sequencing[2]. It is hard to underestimate the impact this has on our everyday lives, with almost every user interface we interact with relying on the underlying technology.

As the demand for more immersive and realistic graphics has surged, so has the power of Graphical Processing Units (GPUs), growing faster than any other microchip found in a PC[3]. Modern GPUs are capable of rendering photorealistic scenes in real time. This technology has made its way into Hollywood, with Disney opting to shoot recent shows against entirely computer generated backdrops. In using a full 3D render, they are able to track the position of the camera and update the backdrop in real time to accurately represent the perspective [4]. Whilst these powerful GPUs have enabled extraordinary possibilities, they have come with added complexity. Through necessity, a layer of abstraction is used between developers and the hardware they work with. High-level APIs such as OpenGL and DirectX offer a convenient means of interacting with GPUs, however they serve to mask the underlying fundamentals of 3D rendering. Game developers can be highly experienced in using these APIs, and yet still lack an understanding of the theory behind 3D graphics[5].

Over the past decade, the way we interact with technology has changed dramatically. The advent of "internet of things" devices has surrounded us with low power embedded systems. These resource limited devices are everywhere, from smart fridges to wearable technology.

This forms the justification for this proposed project. Our goal is to create a 3D graphics library with embedded devices in mind. This will enable developers to harness 3D graphics in resource limited situations, making it possible to render 3D content on a wide range of IOT and embedded devices, enabling alternative means to communicate with a user.

3 Background

Most modern graphical applications rely heavily on Application Programming Interfaces (APIs) to interact with the underlying hardware[6]. A graphical API provides a standardized means for a developer to create graphical applications. These allow a developer to interact with the underlying hardware from a high-level context. As these specifications do not include implementation details, it is up to vendors to create drivers to provide this functionality. One popular API is Vulkan, the successor to OpenGL. This has wide support access platforms, with drivers provided by a range of manufacturers[7]. These cross-platform specifications allow a developer to access hardware-specific optimisations through the high-level API without needing to know the implementation details.

Unfortunately, many common implementations of such API's exist as a "black box" with some users having no knowledge of the core concepts in 3D rendering needed for these to function[8]. Microsoft's DirectX API and Apple's Metal are both proprietary technologies, meaning the code which implements these is hidden from a developer. In using a closed source license, examining the underlying implementation is prohibited, limiting both the extensibility of the product as well as its use as a learning tool.

Graphical programming for embedded applications brings a unique set of challenges. Systems with limited resources may struggle to run a full-featured API. Additionally, devices can range from sensors, wearables to medical instruments, bringing a diverse set of hardware configurations. This diversity necessitates a flexible and adaptable library, capable of supporting the variety of display types, memory and hardware performance.

4 Aims

This project aims to address the challenges faced by developers in creating 3D graphics for embedded systems. The primary goal is to create an open-source graphics library. This will be be designed for resource-limited devices, providing efficient rendering capabilities. Developers will be able to use this library to create visualizations without the need for an existing graphical API. The project will include functionality to load, transform and manipulate 3D objects in a scene, producing a 2D render which can be displayed to a user. Modularity is a key part of the code design, a developer should easily be able to adapt the library to suit their specific requirements and hardware configurations. To showcase the utility of this library, the project will include the development of several example applications.

5 Methodology

The decision to complete this project using the C programming language was made based on several factors. As the project targets embedded systems, performance is a key consideration. C++ is known for its efficiency and low level control making it another possible candidate. The object oriented designed patterns offered are well suited for this project, however this abstraction comes with additional performance overhead[9]. Additionally, C is highly portable, with near universal support across platforms. C++ however can have varying levels of language support across embedded systems. Another benefit of C over C++ is the size of the final binary. This could be a concern for embedded systems with limited storage capacity.

This project has been divided into a set of discrete modules. In defining each step of the rendering pipeline with a set of public data structures and functions, it will be possible for a developer to examine and adapt each module. An example of the benefits this offers would be for a device that uses an unusual display mechanism. If a product requires the final render to be pushed to a grid of LEDs for example, a developer must only replace the display module, without affecting the wider codebase. Another benefit of this decision will be in educational institutions. A student wishing to gain an understanding of 3D graphics will be able to use this utility in their learning. Writing their own implementation for individual steps in the rendering process would be possible without the need of working from the ground up.

Each of the following will be created in the course of this project.

Rasterizing 2D vectors

The initial task will involve creating a module which handles the 2D image pane in memory. This will involve creating a set of functions which take coordinates and colour data for a 2D shape which is then drawn to the pane. These will then be processed and drawn onto the pane.

Display

This section will handle displaying the rendered image to the user. Functions will be created which can initialise a window and update it with data from the rendered pane. The intended implementation is a X11 GUI window to display the pane. Other possibilities include exporting the render to a media file or a text based view for use in a terminal only system. To export a still frame, the portable pixmap format(PPM) could be used. If video export probes possible in the projects time frame, this can be created using multiple PPM frames. A tool such as ffmpeg can be used to combine these into a more common video format.

Object Storage

This module will be responsible for the storage and management of objects in the scene. It must contain functionality to add, update and remove objects from the scene. A mechanism to interact with non-volatile storage will add to the utility of the Additionally, it can be extended with mechanisms to interact with non volatile storage. Formats such as OBJ or FTX will be parsed to extract the necessary information including vertex and pane data.

Object manipulation

This module will provide functionality to manipulate objects in 3D space. Scaling, rotation and transformation around an arbitrary point will all be included. To created these features, it is necessary to conduct research on the process of performing matrix operations and efficient algorithms for their execution. Each of the required manipulations can then be created through applying these matrix calculations to the target object.

Projecting the scene to a plane

Creating a 2D view of the 3D plane requires several processing steps. Firstly, hidden surfaces must be identified to prevent these being added to the render. Next, an algorithm must be used to convert the scene into 2D vectors which can be sent to the drawing module. This will be implemented using an orthogonal projection, in which only the x and y dimensions are considered. If I am able to extend this project further, a complete projection view could be created. This will require much more processing to produce a view accurate to that produced by a camera in the scene.

Example applications

To demonstrate the utility of the final product, a series of example applications will be created. Possibilities for these include; An application which produces a rotating animation of the Utah teapot, demonstrating both loading an object from a file and use of the provided transformations. An interactive application which allows the user to move a camera through a scene. An application for data visualisation using 3D shapes, showing the use case for information communication on IOT devices.

6 Timeline

This project will begin in October 2023 and continue till March 2024. The timeline for it's development has been detailed in the gantt chart below.

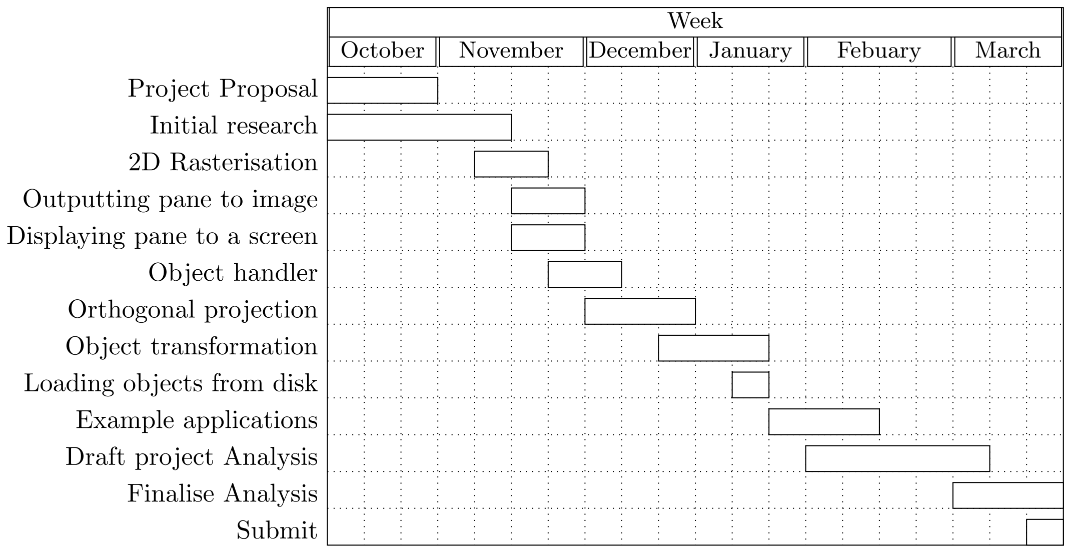


Figure 1: Gantt Chart

7 References

References

[1] Steffan Powell. The uk video games market is worth a record £7.16bn. Accessed Nov 07 2023. `https://www.bbc.co.uk/news/newsbeat-60925567`.

[2] Ping-an He, Dan Li, Yanping Zhang, Xin Wang, and Yuhua Yao. A 3d graphical representation of protein sequences based on the gray code. *Journal of theoretical biology*, 304:81–87, 2012.

[3] Nick Evanson. 25 years later: A brief analysis of gpu processing efficiency. Accessed Oct 23rd 2023. `https://www.techspot.com/article/2008-gpu-efficiency-historical-analysis/`.

[4] Samuel Axon . The mandalorian was shot on a holodeck. Accessed Oct 24 2023. `https://www.unrealengine.com/en-US/blog/forging-new-paths-for-filmmakers-on-the-mandalorian`.

[5] JungHyun Han. *3D graphics for game programming.* Chapman and Hall/CRC, an imprint of Taylor and Francis, Boca Raton, FL, 1st edition. edition, 2011.

[6] Pawel Lapinski. *Vulkan cookbook: work through recipes to unlock the full potential of the next generation graphics API-Vulkan.* PACKT Publishing, 1st ed. edition, 2017.

[7] Vulkan gpu hardware support. Accessed Oct 26rd 2023. `https://vulkan.gpuinfo.org/`.

[8] John F. Hughes et al. *Computer Graphics: Principles and Practice.* Addison-Wesley Professional, 3rd edition edition, 2013.

[9] Michael Barr. *Programming embedded systems : with C and GNU development tools.* O'Reilly, Beijing, 2nd ed. edition, 2006.

Homogenous transformations

Figure 10: Translation matrix

$$\begin{pmatrix} 1 & 0 & 0 & \text{xTranslation} \\ 0 & 1 & 0 & \text{yTranslation} \\ 0 & 0 & 1 & \text{zTranslation} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 11: Euler rotations

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Pitch (x-axis) Yaw (y-axis) Roll (z-axis)

Figure 12: Scale matrix

$$\begin{pmatrix} \text{X scale} & 0 & 0 & 0 \\ 0 & \text{Y scale} & 0 & 0 \\ 0 & 0 & \text{Z scale} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matrix multiplication assembly code

Listing 1: Assembly code for matrix multiplication.

```
multMatrix4Matrix4(matrix4 a, matrix4 b):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-24], rdi
        mov     rax, QWORD PTR [rbp-24]
        pxor    xmm0, xmm0
        movups  XMMWORD PTR [rax], xmm0
        movups  XMMWORD PTR [rax+16], xmm0
        movups  XMMWORD PTR [rax+32], xmm0
        movups  XMMWORD PTR [rax+48], xmm0
        mov     DWORD PTR [rbp-4], 0
        jmp     .L2
.L7:
        mov     DWORD PTR [rbp-8], 0
        jmp     .L3
.L6:
        mov     DWORD PTR [rbp-12], 0
        jmp     .L4
.L5:
        mov     rax, QWORD PTR [rbp-24]
        mov     edx, DWORD PTR [rbp-8]
        movsx   rdx, edx
        mov     ecx, DWORD PTR [rbp-4]
        movsx   rcx, ecx
        sal     rcx, 2
        add     rdx, rcx
        movss   xmm1, DWORD PTR [rax+rdx*4]
        mov     eax, DWORD PTR [rbp-12]
        cdqe
        mov     edx, DWORD PTR [rbp-4]
        movsx   rdx, edx
        sal     rdx, 2
        add     rax, rdx
        movss   xmm2, DWORD PTR [rbp+16+rax*4]
        mov     eax, DWORD PTR [rbp-8]
        cdqe
        mov     edx, DWORD PTR [rbp-12]
        movsx   rdx, edx
        sal     rdx, 2
        add     rax, rdx
        movss   xmm0, DWORD PTR [rbp+80+rax*4]
        mulss   xmm0, xmm2
        addss   xmm0, xmm1
        mov     rax, QWORD PTR [rbp-24]
        mov     edx, DWORD PTR [rbp-8]
        movsx   rdx, edx
        mov     ecx, DWORD PTR [rbp-4]
        movsx   rcx, ecx
        sal     rcx, 2
        add     rdx, rcx
        movss   DWORD PTR [rax+rdx*4], xmm0
        add     DWORD PTR [rbp-12], 1
.L4:
        cmp     DWORD PTR [rbp-12], 2
        jle     .L5
        add     DWORD PTR [rbp-8], 1
.L3:
        cmp     DWORD PTR [rbp-8], 2
        jle     .L6
        add     DWORD PTR [rbp-4], 1
.L2:
        cmp     DWORD PTR [rbp-4], 2
        jle     .L7
        nop
        mov     rax, QWORD PTR [rbp-24]
        pop     rbp
        ret
```