

Games and Graphics assignment

by Michal Pfeiffer

Submission date: 29-Nov-2024 09:23PM (UTC+0000)

Submission ID: 244168672

File name: 40136_Michal_Pfeiffer_Games_and_Graphics_assignment_628112_1431825086.docx (687.83K)

Word count: 1845

Character count: 9094

Application of Graphics Principles Assignment

Games and Graphics Programming

By Michal Emil Pfeiffer (2204569) - BSc (Game Development)



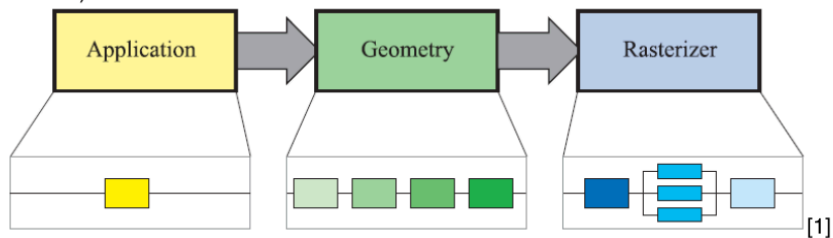
- Task 1 2
 - Graphics Pipeline 2
 - Application Stage 2
 - Geometry Stage..... 3
 - Model & View Transformation 3
 - Shading 3
 - Projection 4
 - Clipping 5
 - Screen Mapping 6
 - Rasterization Stage 6
- Task 2 7
- Task 3 8
- References 9
- Appendix 10

Task 1

Graphics Pipeline

The graphics pipeline can quickly be summarised into three main stages: Application, Geometry and Rasterization [1][3].

The stages will be described in their own sections below (refer to table of contents if needed).



Application Stage

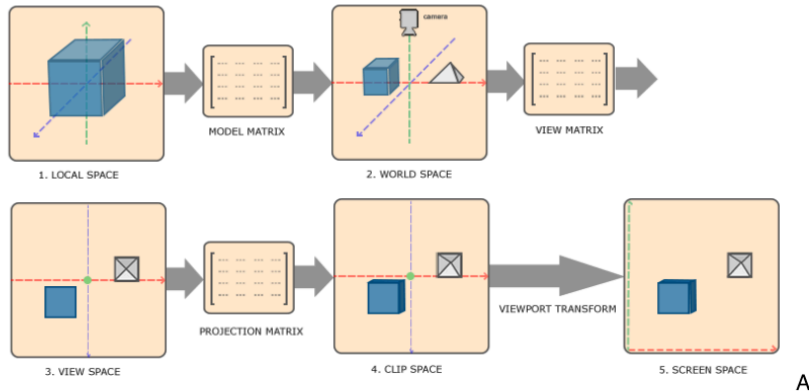
In this stage of the pipeline, everything here is executed in the software [3]. As the application stage doesn't include rendering, most of the processes happen on the CPU cores [1], like preparing the scene, the simulation data, memory allocation, rendering data etc. [1][3] before it is sent towards the GPU to carry out those processes.

This is also the first point where bottlenecks can occur;

- CPU
 - Depending on the generation of the CPU, multi-threading is possible, making it possible to have the process be parallelised, allowing for multiple processes to be executed alongside each other [1][3], which in turn reduces the time needed for a process to wait for its turn to be executed.
- Memory
 - Memory allocation, depending on how it is handled, can affect the next stage being the Geometry Stage, as it can and will affect the overall performance of it [1], like wasting time and space in memory and can possibly lead memory leaks than can break the code.

Geometry Stage

The geometry stage mainly works on vertex and polygon operations and can be summed up into five stages [1][3]; Model & View Transformation, Shading, Projection, Clipping, Screen Mapping.



more visual representation of what happens during the geometry stage [4].

Model & View Transformation

This is the first step vertices and polygons go through, as when an object is being added or created for a scene, the scene also defines a camera [1][3] for which objects (made up of vertices and polygons) will be rendered in, the camera has a position and a direction to view the scene [1][3], and for objects to be viewed in the scene, need to be transformed from local space into world space using multiple transformations, in this stage, a model and view matrix is applied to the object [4].

The model matrix is used to transform an object from local space into world space [4], and the view matrix is used to transform an object from world space into a view space [4], which sets it up for projection later.

Shading

As the name implies, it applies a shade/shadow on objects in view of which the shading is just applying a texture onto the object based on how much light it receives from a light

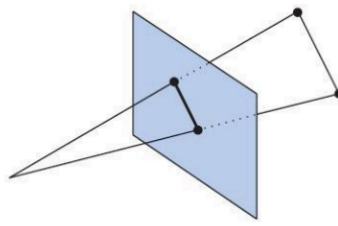
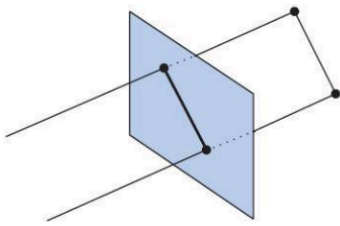
source(s) [1][3], being either ambient light (equal light levels onto each surface) or directional light (stronger/darker shadows on areas with less light with brighter sides where light is pointing at) [3].

The shading of object can also impact the performance depending on when the shading is applied onto the object [1], as depending on which stages have or haven't been completed, will require a different amount of computational power to render the object as it is desired [1].

Projection

This stage is putting the objects into a proper view volume of a cube [1][3], with corner coordinates ranging from $(-1, -1, -1)$ to $(1, 1, 1)$ [3], objects in this view volume are transformed using a projection matrix. The proper rendering step is to apply the 'model', 'view' and 'projection' matrix to each vertex [3] together rather than separately.

Once that is applied, there are two ways an object can be viewed in, being 'Orthographic' or 'Perspective'.



The

two projection methods, Orthographic (left) and Perspective (right) [1].



The key difference between the two projections is that Orthographic is a rectangular view volume (lacks a horizon, faces/sides are in parallel to each other) whereas Perspective is a Frustum view model (has a horizon, has a sense of depth) [1], the depth of the object is stored in a Z-buffer [1][3] (also known as a depth buffer [4]) which is used later in the rasterizing process.

Clipping

After the three projections are applied onto an object/vertex, it must be decided if it is located within the view volume or not, the view volume is the same as the previous view volume, as objects or vertices are only clipped if they are located outside of the volume [1][3]. Once the object is clipped, it is then provided new vertices on where it has been clipped [1] (so that a shader can still be properly applied).

Screen Mapping

Screen Mapping [1] or Projection to Screen Space [3] is further transforming the object from normalised device coordinates to screen coordinates [1], which is done by the viewport transformation, which is done by scaling and then translating the objects [1][3], of which the viewport is the dimensions of the window of which the objects will be displayed in [1]. These calculations are executed within the vertex shader [3], which is provided in older version of OpenGL and DirectX [3], but from DirectX version 10 and OpenGL version 4 onwards, this shader is now user defined [3].

Rasterization Stage

In this stage, the object gets its colour and has pixels assigned to the object so it can be viewed [1][3].

The process covers an area the triangular surfaces are in in dots/pixels, if the triangle covers most of the pixel(s), it is then fragmented [1][3]. This fragment goes through a process called fragment shading [1][3], where it receives data from the shader to be applied onto the fragments, which can be colour and/or lighting of the pixel(s).

Lastly, the fragments are then merged to create an object (that it originally was) to be displayed onto the screen, the merging process consists of smaller processes, like running the fragments through tests like raster operations (testing depth, stencil, colour and alpha blending on the frame-buffer [2]) [1] to then be stored in the colour buffer [1][3]. Once the tests are completed, the fragments are run through the Z-buffer to know which pixels are overlapping (mainly if the pixel is in front or behind the fragment [1][4]), so that it knows which pixels to draw.

Once it all has been finished, the object will be then drawn and displayed on the screen.

There are three main bottlenecks during this stage;

- Raster Operations
 - Raster operations as mentioned above, test depth, stencil, colour and alpha blending which is all done of the frame buffer bandwidth. The best course of action to see if this is the bottleneck is by changing the bit-depth from 32-bit to 16-bit and seeing if that affects the performance [2].
- Texture Bandwidth
 - Texture bandwidth is related to the texture size that needs to be fetched by the GPU, changing the level of detail (also known as LOD) may boost performance if this is a bottleneck [2].

- Fragment Shading
 - Fragment shading mainly refers to the generation of fragments with their colour and depth values, which is inherently a function towards the screen resolution. As the bit-depth was tested for raster operations, this leaves with only one option, being changing the actual screen resolution and seeing if that affects performance, as this can be a rare bottleneck that can occur [2].

Task 2

Task 2 tasks me with finding out the distance of a vertex from a plane and stating (with any explanations/assumptions made) if it will or will not be culled.

The equation of $\mathbf{x} \cdot \mathbf{n} + p_0 \cdot \mathbf{n} = 0$ provides us the distance (mainly to see if the point is on or off the plane) of a point $\mathbf{x}$ from the plane like so:

$$(2, 2, -2) \cdot (1, 2, 0) + (2, 2.5, 3) \cdot (1, 2, 0) = 0$$

$$(2 + 4 + 0) + (2 + 5 + 0) = 0$$

$$6 + 7 = 0$$

$$13 = 0$$

As the value equals to **13** and not **0**, the point is not on the plane and the point is above the plane due to the value being a positive.

Now for any assumptions and explanations to go with them:

1. As the plane normal has only positive values of (1, 2, 0), it indicates the perpendicular direction of where the plane is facing, this being the front or top of the face. If the positive side is the front, then the vertex will be in front of the plane.
2. Assuming that the plane was to be rendered, and the point were to be **behind** the plane, the vertex would be culled if depth testing is enabled. As small objects hidden behind larger objects (or ones that are closer to the camera) don't need to be rendered, therefore culled.
3. Location of the camera matters if the vertex is to be culled or not, as this how we draw a view volume, or a frustum, for objects to be rendered in. As the point itself is far away enough from the camera, it will be rendered in and not culled.



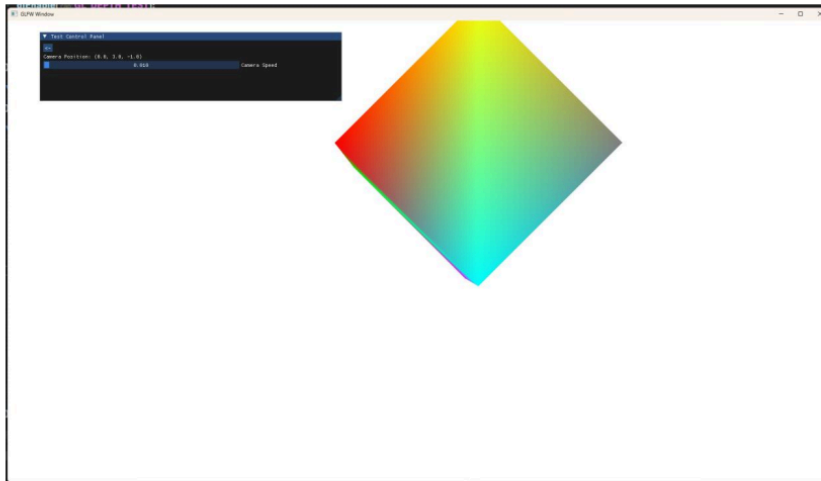
Task 3

The code is mainly written to test for Task 1, if the translation is to be ignored, it can be tested for Task 2, to see if it is far enough away to be rendered.

The model matrix is formed as such:

```
model = translation1; //Setting the model to the specified vertex  
model = rotation * translation2 * model; //Translating and then rotating the vertex (as matrices go from right to left)
```

This may or may not be wrong, but with this implementation, it is shown as such:

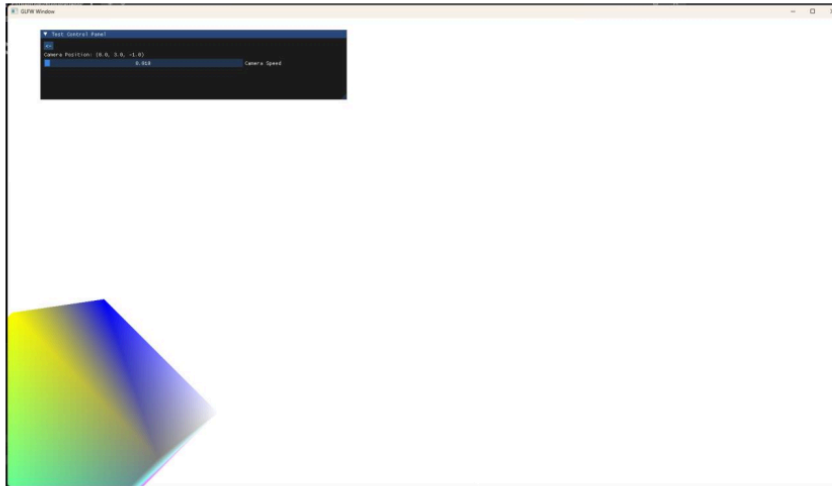


The center of the cube is clearly visible and rendered.

If it were to be implemented as such:

```
model = translation1; //Setting the model to the specified vertex  
model = model * translation2 * rotation; //Translating and then rotating the vertex (as matrices go from right to left)
```

If the calculation is done in this order, it will be rendered as such:



As you can see, the center of the cube is in a different location, but the center of cube is still visible and rendered.

The code will be uploaded with the iteration that was shown first.

References

- [1]
A. Gkoutis, "Overview of the Graphics Pipeline," 2019, [Online]. Available:
<http://www.fragmentstorm.com/overview-of-the-graphics-pipeline>. [Accessed: 27-Nov-2024]
- [2]
C. Cebenoyan, "Chapter 28. Graphics Pipeline Performance," N/A. [Online]. Available:
<https://developer.nvidia.com/gpugems/gpugems/part-v-performance-and-practicalities/chapter-28-graphics-pipeline-performance>. [Accessed: 27-Nov-2024]
- [3]
Imperial College London, "Graphics pipeline," pp. 1–5, N/A [Online]. Available:
<https://www.doc.ic.ac.uk/~bkainz/graphics/notes/GraphicsNotes0405.pdf>. [Accessed: 27-Nov-2024]
- [4]
Learn OpenGL, "LearnOpenGL - Coordinate Systems," 2014 [Online]. Available:
<https://learnopengl.com/Getting-started/Coordinate-Systems>. [Accessed: 27-Nov-2024]
- [5]
D. Nikos and M. Ross, "18.1.2.3 Rasterization Bottlenecks," Jan. 2001 [Online]. Available:
<http://www.bluevoid.com/opengl/sig00/advanced00/notes/node358.html>. [Accessed: Nov. 27, 2024]
- [6]

The Organic Chemistry Tutor, "How To Find The Distance Between a Point and a Plane," Dec. 17, 2019
[Online]. Available: <https://www.youtube.com/watch?v=zWMTTRJ0l4w>. [Accessed: Nov. 29, 2024]

Appendix

These are the values that I have been provided in the appendix of the assignment (the values are here for easier access when working):

Vertex	Rotation Angle	Translation			
(2, 2, -2)	45.0°	(2, 0, -1)			
Viewpoint	Up Vector	Look At Point	Field of view	Near plane distance	Far plane distance
(0, 3, -1)	(0, 1, 0)	(1.5, 0, 0)	36.0°	2	70

The code will be uploaded in a **.zip** format, the code is what we have worked on in the lectures.

Games and Graphics assignment

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

71 /100

Engagement with Literature: 12/15

Knowledge and Understanding: 15/20

Cognitive and Intellectual Skills: 20/30

Practical Skills: 20/30

Transferable Skills: 4/5

Overall 71/100

Comment:

A great overview of the graphics pipeline.

Missing the demonstrations of the required transformations.

Avoid bullet points where possible.

screenshots should be marked as figures with captions, and then a table of figures provided under the table of contents, to allow for easier navigation for the reader and a more professional flow.

page numbers/footer

See inline comments also

PAGE 1



Comment 1

A cover page can take up an entire page!

Dont be shy to use an extra page.



Comment 2

ideal thing to reference!



Comment 3

This would have been better to layout as a CPU as a heading and the text as a paragraph below rather than using the bulletpoints.



Comment 4

A nice comparison example



Comment 5

Good good