



CM3203: One Semester Individual Project

Final Report

Efficient Ordered Transparency Rendering for Static 3D Meshes

Conor McSweeney (1834704)

Supervised by Dr. Xianfang Sun

2021-05-13

Abstract

This project aims to design a novel method for rendering transparent 3D meshes with correct depth ordering more efficiently than existing approaches, in the special case where the mesh is static. Static transparent meshes occur frequently in applications such as CAD software. By taking advantage of the static nature of these meshes, a pre-processing step may analyze and re-organize mesh data in such a way that it can be rendered from arbitrary viewpoints in real-time, with minimal real-time computation. The project then goes on to produce a simple interactive graphical application, for demonstrating the novel method and evaluating it by comparing against a selection of existing methods. The application should further allow the evaluation to be performed on multiple hardware environments including mobile devices.

Acknowledgements

I would like to thank Dr. Xianfang Sun for supervising this project.

I would also like to thank Morgan McGuire and Louis Bavoil for publishing in great detail their works on Weighted Blended Order-Independent Transparency, and in general their contributions to the field of computer graphics.

Free 3D models used in this project were sourced from the Blender Foundation and the University of Utah.

Table of Contents

Efficient Ordered Transparency Rendering for Static 3D Meshes.....	1
Abstract	2
Acknowledgements	2
Table of Contents.....	3
1. Introduction	5
1.1. Project Aims and Deliverables.....	5
1.2. Personal Aims	5
2. Background	6
2.1. Principles of Transparent Rendering	6
2.2. Example Problem Case	6
2.3. Existing Work	6
2.4. Reflection on Existing Work	7
2.5. OpenGL ES and WebGL	7
2.6. Index Buffers	8
3. Algorithm Description	9
3.1. Exploitable Properties in One-Dimensional Case	9
3.2. Expanding to Multiple Dimensions	10
3.3. Structure of Implementation	11
3.4. Potential Optimization by Culling.....	12
3.5. Applying to a Mesh in a Scene	13
3.6. Self-Intersection Problem	13
4. Implementation of Application and Graphics Framework	14
4.1. Application Structure	14
4.2. User Interaction via Camera Control	14
4.3. Control Menu	15
4.4. Statistics	15
4.5. External Libraries	15
4.6. Object Models and Shader Programs.....	15
4.7. Common Object Shading.....	16
5. Implementation of Transparency Methods	17
5.1. Naive Method - No Depth Testing.....	17
5.2. Triangle Sorting	17
5.3. Weighted Blended Order-Independent Transparency	19
5.4. Novel Method - Multiple Clipped Fixed Ordering	20
5.4.1. Pre-computation of Fixed Orders	20

5.4.2. Multiple Pass Rendering with Clipping	21
6. Results and Evaluation	24
6.1. Evaluation Procedure for Transparency Methods.....	24
6.2. Evaluation of Method Visual Quality and Applicability	24
6.3. Evaluation of Method Performance on Multiple Systems	29
6.3.1. High-end Laptop Computer	29
6.3.2. Performance-limited Laptop Computer	30
6.3.3. Low-end Laptop Computer	31
6.3.4. Smartphone	31
6.4. Overall Evaluation	32
7. Future Work.....	33
8. Conclusions	33
9. Reflection on Learning	34
Appendices	35
Work Listing by Week	35
Running the Application.....	36
References	37

1. Introduction

This report assumes the reader is familiar with basic concepts and terminology of rasterized 3D graphics, including for example meshes and shaders. It will not require specific knowledge of transparency rendering or the names and operations of particular graphics API functions.

1.1. Project Aims and Deliverables

In this project I aim to develop and evaluate a novel method for correctly rendering static transparent meshes in a rasterized 3D graphics pipeline, for example OpenGL.

In comparison with more photorealistic rendering techniques like ray tracing, transparency is typically not handled well in rasterized graphics pipelines. This is due to the fundamental operation of this type of rendering, where the process and order of drawing a scene is not implicitly related to its contents and structure, or to the forms and materials of the objects within the scene.

The project will investigate how the properties of a static mesh in particular may be exploited to perform this task more efficiently than existing general-case methods, intended for dynamic scenes. The performance and applicability of the novel method will then be evaluated in comparison with these existing methods, by implementing several of the methods in an interactive demonstrational application.

The application will be developed in WebGL, a wrapper for the OpenGL ES API that runs in modern web browsers. This will allow the evaluation of the novel method to include both high-end (desktop) and low-end (mobile, embedded) devices within the same implementation. The final delivered application will be capable of displaying a scene consisting of multiple transparent objects from arbitrary viewpoints and will allow the user to select and compare the various methods to be evaluated, including the novel method. It will also provide quantitative measures of performance by timing both the CPU drawing loop and the GPU render processes.

1.2. Personal Aims

By completing this project, I aim to expand my own skills within 3D computer graphics, as well as potentially contribute to the ongoing research in this area. The topic of transparency rendering and its related research is also something that I have formed a personal interest towards in the past few years. I have become aware of the challenges and issues of this topic through development of 3D applications, and also through using applications and games where an attention to detail makes these issues obvious.

2. Background

2.1. Principles of Transparent Rendering

Rasterized 3D graphics pipelines are highly optimized for unordered opaque rendering, utilizing a depth buffer to allow new fragments to be fully obscured by previously drawn fragments that are closer to the viewpoint. This implements a basic binary visibility system; A fragment is either fully visible or fully obscured.

However, this system does not always work when it is applied to transparent rendering. Typically, transparent fragments are rendered using some "alpha" opacity value to blend with their background, which determines how much the background color is attenuated and how much the new fragment's color contributes to the result. These fragments must be explicitly drawn in the correct order from furthest to nearest in the scene to produce a visually correct output, as this typical blending function is non-commutative, i.e., a sequence of fragment blends applied to a pixel will not produce the same result if it is rearranged.

In the past, real-time applications have often been forced to ignore or work around this issue. This may include limiting use of transparency to a single layer, using exclusively a blending function that is commutative such as additive (and subtractive) blending, or simply ignoring the issue entirely resulting in obvious visual artifacts. However, there has been a large amount of research with the goal of finding better solutions, many of which have become more viable for real-time applications on typical consumer hardware in the last few years.

2.2. Example Problem Case

One notable project where obvious visual artifacts have been prevalent for a long time is the popular sandbox video game "*Minecraft*" developed by Mojang Studios. In its current state, the game contains many transparent elements which may be placed arbitrarily in the game's environment, either by players or by its procedural generation features. As such, it has suffered many issues with attempting to correctly render these elements over its twelve-year (at time of writing) lifetime. The currently employed solution involves periodically sorting transparent faces of relatively static environmental elements, while no attempt is made to correctly sort dynamic elements and effects due to performance concerns.

The issues caused by this approach were first formally reported on the game's public issue tracking system in early 2013 (issue number MC-9553). Since then, there have been continuous updates to the issue report. The prevalence of the problem and many cases in which it arises eventually lead to this issue being closed and split into many smaller reports in late 2019. There is still ongoing development in 2021 which is attempting to solve the problem by implementing one of the existing approaches mentioned in Section 2.3.

2.3. Existing Work

Existing approaches to this problem fall into two main categories. The first category aims to achieve an exact solution, by explicitly or implicitly sorting transparent fragments. This category typically suffers from requiring significant processing power, memory capacity, and/or bandwidth. For less complex geometry, a simple method based on sorting of primitives (triangles) on the CPU may produce equivalent results to methods in this first category.

A popular example in this category is Dual Depth Peeling (Bavoil and Myers 2008). This is an improvement on a prior method called Depth Peeling. In Depth Peeling, the transparent elements of scene are rendered in their entirety in multiple passes, with each pass recording one new layer of transparency. In this method, the maximum number of transparent layers at any given screen position is equal to the number of complete render passes, and as such is highly limited by bandwidth and fill rate. Dual Depth Peeling doubles the number of transparent layers achievable by a given number of passes, peeling two layers at each pass; one from the front and one from the back.

Another example is Per-Pixel Linked Lists of fragments (Yang et al. 2010). This method works by building real-time linked lists of transparent fragments per pixel, containing fragment depth values, effectively storing three-dimensional information on transparent layers. After the complete scene has been rendered in this way, each pixel's linked list is composited in depth order to resolve a final output color. This method is limited by memory capacity, as it has unbounded memory usage as the linked lists dynamically grow. It is also limited by bandwidth and processing power as the lists are sorted and composited in the final resolve pass.

The second category merely tries to approximate a solution, which may visually differ from an exact "ground truth", while still being visually acceptable in a given context. These methods typically require some tuning in order to be visually acceptable.

An example in this category is Weighted, Blended Order Independent Transparency (McGuire and Bavoil 2013). This method is based on a weighted average, where the weighting function determines the weight of a given transparent fragment based on its depth and alpha value. It is able to loosely approximate a visibility function, that is the relative amount of light visible from a given fragment or layer. The weighting function works optimally when selected and tuned for the specific scene it will be applied to, as the approximation does not hold for wide ranges of depths and alpha values.

2.4. Reflection on Existing Work

A common feature between all the methods in these categories is that they aim to provide a general solution, operating on an entire scene regardless of how its contents are stored, organized, and potentially animated. However, a popular test case used to demonstrate these methods is a single static mesh with no animation, where only the transformation, viewpoint and shading may change. While this does not showcase the full capabilities of those methods, it does correspond to a common scenario encountered in real life applications like CAD tools, games, and scientific visualizations. This project aims to produce a more specialized method which can be much more optimized for these static mesh use cases.

2.5. OpenGL ES and WebGL

For 3D graphics applications where performance is highly important, especially those developed for research purposes, it is common to use a desktop graphics API such as OpenGL and to develop the application as a native program (in C or C++). Using a recent version of the desktop OpenGL API allows access to advanced "modern" graphics pipeline features, such as compute shaders and functions designed to optimize performance over previous versions, as well as greater flexibility with resources and pipeline customization. However, for this project these features were determined to be unnecessary.

OpenGL ES is a "well-defined subset of desktop OpenGL" (Khronos Group 2015) designed primarily to run on embedded systems including game consoles, smartphones, and tablets. It provides fewer modern features than desktop OpenGL, lacks some advanced functionality, and has more restricted resource capabilities (e.g., a smaller number of available texture units). These constraints allow the API to be highly cross-platform, achieving satisfactory hardware-accelerated performance on many devices, with no dependency on platform-specific features.

WebGL is a web-browser-based graphics API which takes advantage of the cross-platform capabilities of OpenGL ES, allowing 3D graphics to be used as a native component in modern web browsers on many different devices, without requiring any extensions or additional applications. The API is accessed through JavaScript, the portable scripting language used in web pages. While JavaScript performance is notably lower than that of native applications, even with the optimized script engines provided by modern browsers, most CPU-bound operations in a graphical application achieve adequate performance. This also does not affect the performance of GPU-bound (hardware accelerated) graphics processing operations.

As the advanced features of desktop OpenGL are not required by this project, WebGL presents a suitable graphics API for developing a novel rendering method and evaluating it on multiple platforms.

2.6. Index Buffers

An important feature of modern OpenGL referenced later in this report is the concept of index buffers, often alternatively referred to as element buffers in OpenGL-specific terminology. An index buffer relates the order of drawn vertices, and the resulting construction and order of primitives (typically triangles), to the vertex data for a given mesh. This reduces duplicate vertex data down to a single index value to be duplicated, saving bandwidth and memory usage. A simple example would be to draw a quad. In this case six indices are used to specify two triangles, but vertex data is only required for four vertices. With relevant data in vertices 0, 1, 2 and 3 defined in counter-clockwise order (the standard front-facing winding order in OpenGL), a suitable list of indices would be [0, 1, 2, 0, 2, 3].

The importance of the index buffer concept to this project lies primarily with the ability to specify multiple index buffers for the same mesh and use them arbitrarily. These multiple index buffers can be used to specify a set of rendering orders, either static or dynamic, for the same mesh data.

3. Algorithm Description

3.1. Exploitable Properties in One-Dimensional Case

In order to develop this method, we must consider how the specific properties of a static mesh may be of use in the context of depth ordering.

It is helpful to consider meshes in terms of point clouds, where each point corresponds to the centroid of a triangle. In most cases this will produce high quality results, equivalent to the results of explicit triangle sorting. Cases with more complex geometry, that primarily being self-intersecting meshes, may be handled by programmatically splitting up and organizing the triangles appropriately. This is discussed in further detail in Section 3.6.

Consider the following scenario in Figure 1, showing a set of points constrained to a single dimension. Note that such a one-dimensional scenario is not practical to consider as a rendering problem and may be hard to visualize as such; it is merely used for illustration of important principles before generalizing to higher dimensions. The points are represented in the scene by labelled circles. The viewpoint which those points should be ordered towards is represented by the human figure.

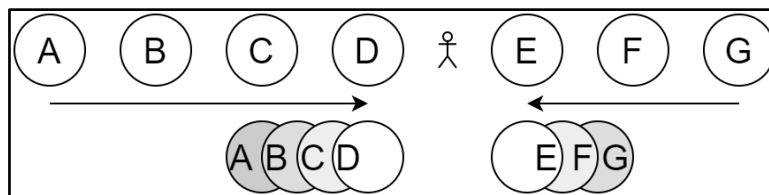


Figure 1: Points to be ordered in one dimension.

It can be trivially seen that ordering the left-most set of points from furthest to nearest relative to the viewpoint would result in a draw order of [A, B, C, D]. For the right-most set the resulting draw order will be [G, F, E]. This is equivalent to reading the left set in a fixed left-to-right order, and the right set in a fixed right-to-left order. The sets are considered to be independent.

An efficient implementation for rendering this scenario is to construct two fixed order lists; one from left to right as [A, B, C, D, E, F, G], and the other from right to left as [G, F, E, D, C, B, A], as in Figure 2. Each order list is rendered in turn, except that points lying outside the valid viewing range are discarded (clipped). The left set renders points [A, B, C, D], discarding further points as they are no longer located to the left of the viewpoint. Likewise, the right set will render points [G, F, E] and discard the remaining points. This is shown in Figure 3.

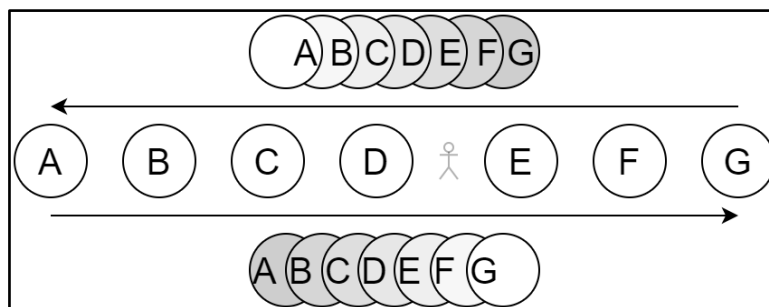


Figure 2: Fixed ordered point sets.

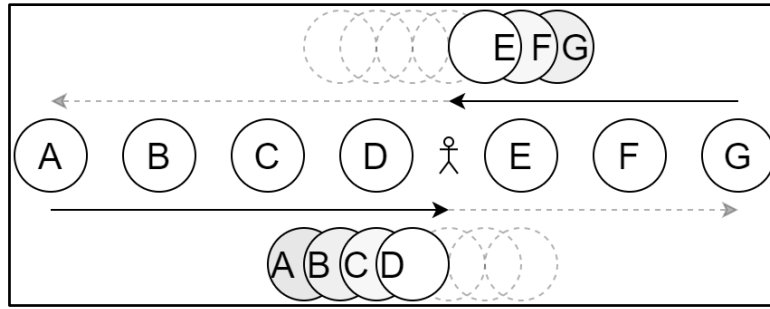


Figure 3: Clipped fixed-order rendering procedure.

3.2. Expanding to Multiple Dimensions

From the above case with only a single dimension, we must now apply this principle to higher dimensions. When using the Euclidean distance metric, it may not be immediately clear how this can be achieved. However, for the purposes of depth sorting, other distance metrics such as Manhattan (taxicab) distance are equally suitable.

By using the Manhattan distance metric instead, we see in Figure 4 how the same principle from the one-dimensional case may be applied to two dimensions by using axis aligned clipping relative to the viewpoint to split the mesh up into spatial regions. From this we can generalize to three-dimensional space or higher. Note that the method may utilize Euclidean distance in a practical implementation, however Manhattan distance will be used for the remainder of this project and report.

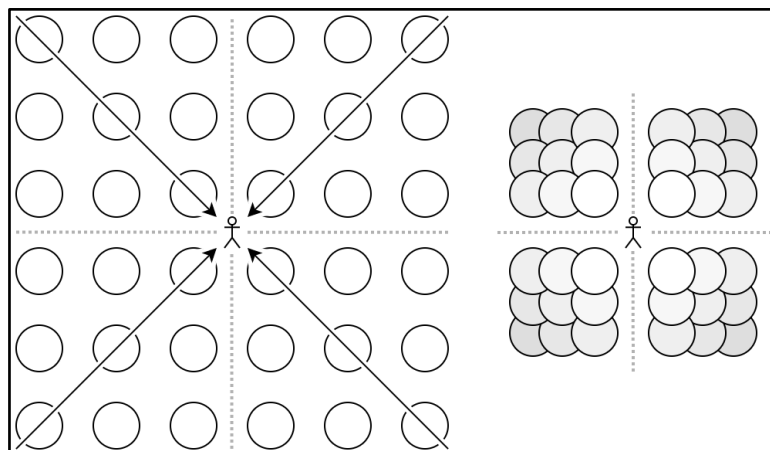


Figure 4: Multiple clipped fixed ordering in two dimensions.

Note that the horizontal and vertical lines shown in Figure 4 and further illustrations of this style represent the clipping planes, or clipping lines in two dimensions, along which the mesh will be clipped to split it into multiple regions. These planes are not necessarily located at the origin in world space or model space; They are centered on the viewpoint which may be arbitrarily positioned. Figure 5 shows how these clipping planes are aligned with an arbitrary viewpoint away from the origin.

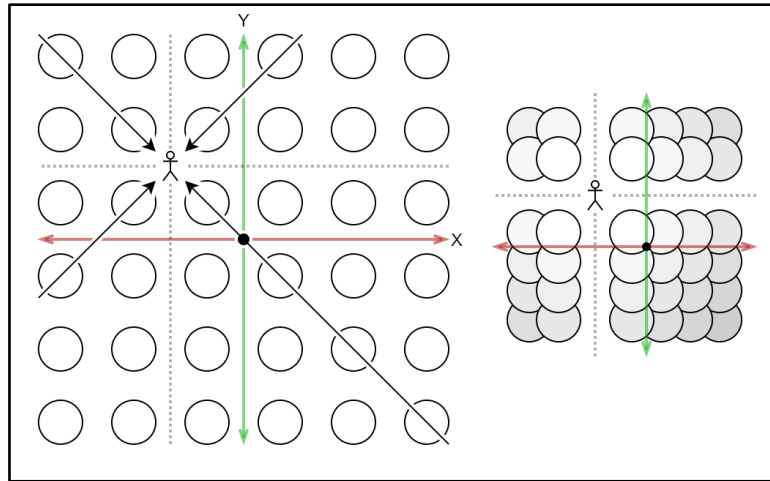


Figure 5: Clipping to a viewpoint not at the origin.

3.3. Structure of Implementation

The method will be implemented in two stages. The first stage will construct the required fixed-order static meshes from the mesh data, and this should take place as soon as the model data is loaded. This stage only needs to operate once at loading, or if the model data is modified, so it does not require an efficient sorting operation other than to reduce the associated loading time.

The second stage uses the above principle in order to draw the mesh in multiple passes. The illustration of the principle in one dimension shows a leftward pass and a rightward pass, and the two-dimensional case shows more clearly with four passes how this corresponds to regions of the model space. In the target three-dimensional case, there will be eight possible passes.

In each pass, there will be three clipping planes set up which are aligned to the axes of the model space. Depending on which pass is currently being drawn, i.e., which spatial region of the mesh should be rendered, these clipping planes will be configured to discard geometry which lies on one side of the plane or the other. The result is that geometry will only be displayed for one of the eight model space regions corresponding to this pass. Each pass will also select the appropriate fixed-order mesh to render. Figure 6 illustrates the two-dimensional equivalent of the overall rendering operation, showing how each pass renders a clipped region of the appropriately ordered mesh, and constructs the whole mesh shape from these multiple orders.

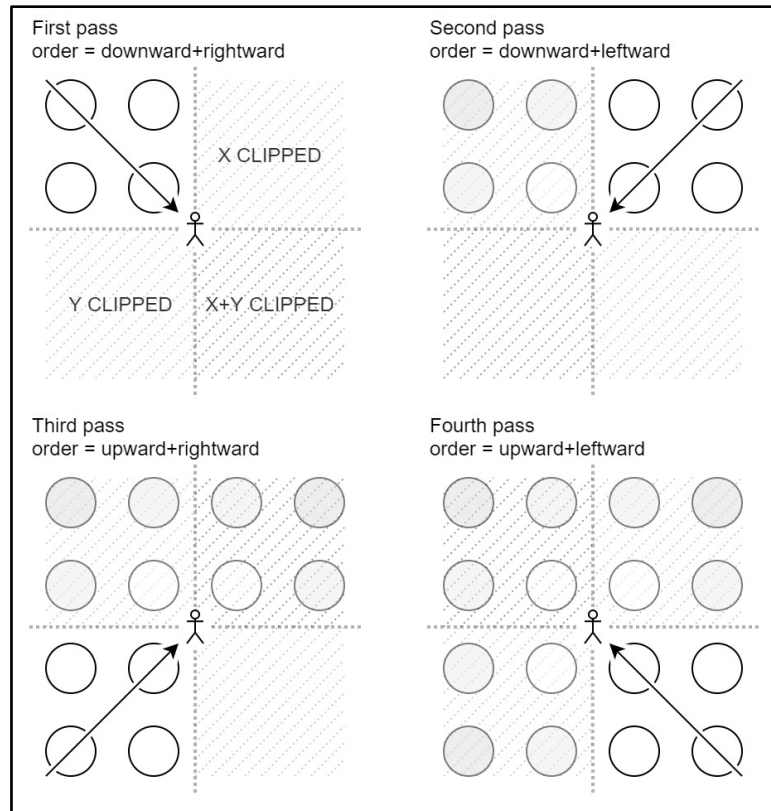


Figure 6: Overview of rendering operation illustrated in 2D.

3.4. Potential Optimization by Culling

Rendering all eight passes is not necessarily required. The algorithm can be optimized by determining which passes are not visible and may be culled (skipped). The primary case for culling a pass is when its corresponding region is completely outside the view volume. In common cases where the field of view does not exceed 90 degrees in any direction (vertical, horizontal, and potentially diagonal), at least four of the eight passes may be culled. If the field of view exceeds 90 degrees, then this cannot be guaranteed. The possible scenarios for this culling condition are shown in the two-dimensional equivalent case in Figure 7 with dashed circles representing culled geometry.

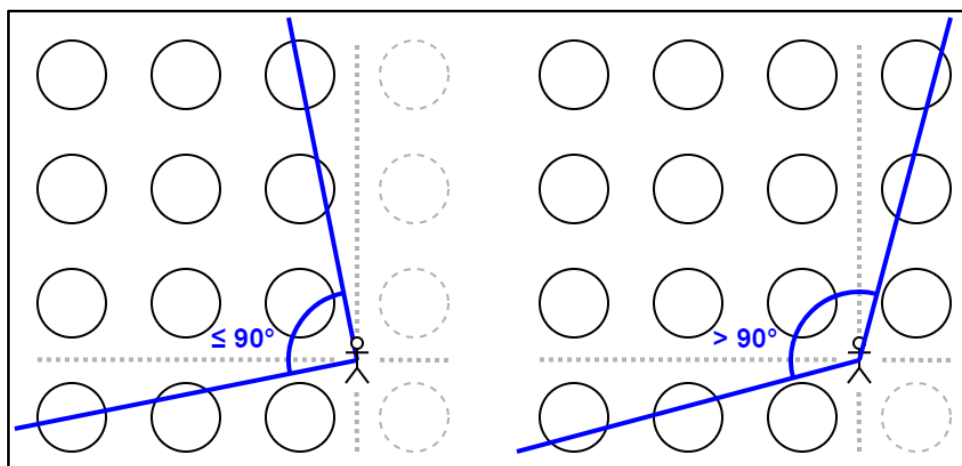


Figure 7: Culling passes based on the view volume.

Another case for pass culling is when a pass is known to be fully outside the mesh, as this will clip the entire mesh and render no new geometry. Rather than checking triangles directly, it can be determined very efficiently whether the pass is fully outside the mesh by only considering the (axis aligned) bounding box of the mesh and comparing it against the viewpoint and the effective region of the pass. Using this culling method, a mesh may be fully rendered in merely one single pass.

3.5. Applying to a Mesh in a Scene

Typically, the object represented by a given mesh will have some transformation placing it within a larger scene, and this will often include some rotation. To correctly handle this, the axis aligned clipping should take place in the model space of the mesh rather than world space, as the fixed order is also computed using model space coordinates. This will require projecting the viewpoint location back into model space.

Another consideration is that treating triangles as points is not suitable for the clipping stage. Instead, clipping must be performed on the resulting fragments. This allows a triangle to be split between multiple clipping regions as appropriate.

It is also worth noting that this method does not resolve mesh self-intersections or inter-object ordering. These issues must be handled at a higher level, for example preventing object intersections and sorting whole objects by their centroids in real-time. This method concerns only intra-object ordering, or self-transparency.

3.6. Self-Intersection Problem

As mentioned earlier, there is a flaw with this method that results from naively treating triangles as points. As with other triangle-based methods such as real-time triangle sorting, self-intersecting triangles or triangles with complex overlapping arrangements will not be handled well and will produce visible artifacts, as there is no defined correct order.

A solution to this that has been explored in the context of real-time sorting is BSP (binary space partitioning) trees. This involves arranging geometry into a spatial tree structure, splitting problematic polygons into two or more parts, and this resolves the mentioned issues.

Use of a BSP tree in this way typically involves traversing the tree to determine a current draw order in real-time. This follows the same performance problems of ordinary triangle sorting. However, if a BSP tree is applied to the newly proposed method, this traversal is again only required during initialization, in place of sorting. For this project, the demonstration of the new method will not implement a BSP tree solution. This is left open to potential future work.

4. Implementation of Application and Graphics Framework

4.1. Application Structure

To demonstrate and evaluate the novel method, a simple graphical framework was created. The framework exists as a web application written in JavaScript and is based on the low-level WebGL 2 API. No existing graphical framework was used in the application, as implementing the novel solution requires low-level control over the rendering pipeline in ways which would be unviable in these frameworks. WebGL and the web application format were chosen to allow the final application to be highly portable, including the ability to run and evaluate it on mobile devices.

The implemented graphical framework allows the application to load and arrange multiple static object models (meshes) into the scene and display them with a fixed shading and colorization, intended to create deliberately high contrast between different faces (triangles) in each object. This helps to highlight the effectiveness and potential artifacts of each implemented method. The framework is also able to load multiple shader programs and use them in different ways within the render loop to implement each transparency method.

4.2. User Interaction via Camera Control

One useful feature of the framework is a simple camera control with multiple modes, using the mouse and keyboard to allow the user to view the scene from different angles and positions. To allow the application to be used on mobile devices, the camera control also accepts touch inputs in most of the control modes. The implemented camera modes are named "Orbit", "Orbit Auto", "Free", and "Test".

The "Orbit" and "Orbit Auto" camera modes lock the camera target to the origin of the scene, allowing the user to orbit around the collection of objects and view them from arbitrary exterior angles, by dragging with the mouse. The camera distance from the origin can be adjusted using the scroll wheel or the W and S keys. The "Orbit Auto" mode additionally slowly revolves the camera around while the user is not dragging the view, automatically showcasing different angles. Touch control in these mode uses one finger to rotate and two fingers moving vertically to adjust distance.

The "Free" mode implements basic free motion control. Similar to the orbiting modes, the user can drag with the mouse to rotate the view. However, rotation takes place relative to the camera, and there is no fixed look target. The camera can be positioned by using the WASD key group to move horizontally relative to the view direction, and the spacebar and shift keys can be used to move upwards and downwards respectively. Keyboard control (positioning) and mouse control (rotation) may be used at the same time. In this mode, touch control uses one finger to rotate and two fingers to pan. Panning operates entirely relative to the view direction.

The final "Test" camera mode causes the camera to rapidly orbit around the scene through a fixed path. It uses a constantly increasing horizontal angle or yaw, combined with a maximally oscillating vertical angle or pitch. This gives a consistent testing scenario for evaluating the performance of the implemented transparency methods, which may vary depending on the viewing angle. No user input is accepted in this mode.

4.3. Control Menu

The framework also provides a simple control menu, allowing the user to select the transparency method to test, the camera control mode as described above, and control the overall alpha value for the objects in the scene. It also records and displays statistics in this menu, showing the mean CPU and GPU processing time per frame, and allowing the user to artificially multiply the rendering workload by a certain factor for evaluation purposes. The menu is implemented with simple scripted HTML elements, as an overlay in the top left corner of the screen. It can be minimized to a single button (to maximize again) allowing the user to see a more complete scene.

4.4. Statistics

The statistics in the menu are updated every 200 milliseconds from data collected in the graphics process. This data consists of mean averages for frame rate, CPU processing time and GPU processing time, computed since page load or the most recent change of settings relevant to performance.

CPU processing time is measured using JavaScript's built-in performance timing functions, and considers the time taken to render the scene only, i.e., graphics API calls and any necessary manipulation of the application state or object meshes. GPU processing time is measured using the WebGL2 disjoint timer query extension, which is not available on all devices. Frame rate is simply computed by the total number of scene frames drawn during the measured time, divided by this time. Processing time is computed similarly, by accumulating totals and dividing by measured time.

4.5. External Libraries

The application makes use of two external libraries for the rendering process. The first library is `webgl-obj-loader` by Aaron Boman (<https://github.com/frenchtoast747/webgl-obj-loader>), which is used to simplify loading 3D models from the open Wavefront OBJ format. The library parses the OBJ plain-text input, triangulates faces and provides usable vertex data to the rest of the application. The second library is `glMatrix` by Brandon Jones (<https://glmatrix.net/>), which implements common matrix and vector math along with classes for easy management of these data structures.

The data for the object models and shader program source code are loaded as JavaScript files containing data strings, statically loaded within the main HTML file along with the main application script. This simplifies development and deployment as opposed to using a resource management system to load data directly from more appropriate file types.

As mentioned above in Section 4.1, the application does not make use of any external library for its graphics framework. The popular library Three.js was considered during the early stages of the project, however it was quickly determined to be unsuitable because of its tight control of the rendering process.

4.6. Object Models and Shader Programs

The models used in the application are a basic sphere, a smoothed Menger sponge, a Utah teapot, and a smoothed version of the Blender Foundation's "Suzanne" or "monkey" test model. These models provide a range of topologies and different levels of complexity. These models are originally stored in Wavefront OBJ format, but their plain-text data is contained as strings in JavaScript files for the sake of simple inclusion from the application's main HTML file, compared to more complex loading processes with further network requests. These JavaScript files are placed in the "model" subdirectory.

The source code for GLSL shader programs is similarly stored as strings in JavaScript files, under the "shader" subdirectory. The vertex shader and fragment shader code for each shader program is stored in the same file. While this JavaScript string encapsulation of the data introduced some difficulty to the development process through string formatting and lack of syntax highlighting, this did not pose any major trouble due to the relatively simple structure of the shader programs used.

4.7. Common Object Shading

The object shading implemented in this application is common between transparency rendering methods. The main shader program for each method makes use of a copy of the common function "getVertexColorPremult" in the corresponding vertex shader, shown in Listing 1 below. As suggested by the function name, shading is performed per vertex for performance reasons. The higher visual quality of per-fragment shading is not necessary for the purpose of this project, that is to simply provide contrast between overlapping geometry.

The function outputs a color in a pre-multiplied format, meaning that the RGB color values contributed by diffuse surface reflection are multiplied by the surface alpha. One reason to use this format is that the blending function does not have to perform this multiplication per fragment later on. Another reason is that, by adding the color values produced by specular reflection without pre-multiplying, the specular highlights will not be blended according to the surface alpha. They will instead appear to be additively blended, which is more physically accurate and therefore visually appealing, without requiring a separate pass or blending function.

The diffuse surface color of each vertex is primarily provided by its position in model space. This adds a high-contrast color gradient across each object, achieving the goal of highlighting the overlaps of transparent faces. It is also affected by the view space surface normal, slightly darkening the edges of objects where faces are oriented away from the viewer. The alpha value or opacity is controlled globally for all objects and faces via a uniform, configurable through the user interface. This allows the user to evaluate the visual quality of each transparency method at different levels of opacity. As detailed in Section 2.3, the quality of the Weighted, Blended Order Independent Transparency method significantly depends on the typical alpha value used.

Listing 1: Common object shading code (in vertex shaders)

```
vec4 getVertexColorPremult(void) {
    // Returns vertex color with alpha, premultiplied (for specular effects)
    // Simple hard-coded shading to add more variation in the mesh
    const vec4 lightVector = normalize(vec4(-1, 3, 1, 0));
    vec3 baseColor = min(max(vec4(aPosition, 1).xyz * 0.5 + 0.5, 0.0), 1.0);
    vec4 normVector = vec4(aNormal, 0);
    vec4 normView = normalize(uMatModelView * normVector);
    float edgeShade = 1.0 - pow(1.0 - abs(normView.z), 2.0) * 0.7 + 0.3;
    vec4 reflectionVector = reflect(lightVector, normVector);
    vec4 reflectionView = normalize(uMatModelView * reflectionVector);
    float specularLight = pow(max(0.0, -reflectionView.z), 30.0) * 0.5;
    return vec4(min(baseColor * edgeShade, 1.0) * uAlpha + specularLight,
                uAlpha);
}
```


5. Implementation of Transparency Methods

The following transparency methods are implemented conditionally in the application within the `"Graphics.prototype.renderScene"` function in `graphics.js`, after clearing the output frame buffer, enabling blending, and disabling depth testing. Prior to this function being called, the camera and associated transformation matrices are updated, and the scene is depth sorted at an object level. Objects are deliberately placed such that they do not intersect or overlap each other, so this is sufficient when combined with a sorting-based method to sort the triangles within each object's mesh.

5.1. Naive Method - No Depth Testing

The default method implemented in the application for reference is to naively draw the transparent meshes with depth testing disabled. Disabling depth testing ensures all "layers" of triangles will be drawn, albeit with an incorrect blending result. As discussed in Section 2.1, this method is useful only when using commutative blending functions or when all transparent geometry to be blended is of the same color. It was implemented primarily to illustrate the clear artifacts in the base case where transparency is not handled correctly.

This method simply iterates over the objects in the scene, applies the object transformation and draws the object's mesh exactly as defined in the loaded model data. For this method, the default shader program is used. This shader simply generates vertex colors as described in Section 4.7, and outputs interpolated colors directly to produce the final fragment color. The output is in pre-multiplied alpha form, so the object must be drawn with the blend function having source factor of 1 (`ONE`) and destination factor of $1 - \alpha$ (`ONE_MINUS_SRC_ALPHA`). Note that the source factor differs from the typical non-pre-multiplied alpha blending function, in which it is given as α (`SRC_ALPHA`). This allows specular highlights to appear independent of surface (diffuse) alpha.

The other methods below are all variations on this basic rendering procedure.

5.2. Triangle Sorting

This method follows largely the same code path as the naïve method above, however the object mesh will be triangle sorted immediately prior to being drawn.

The camera position is transformed into model space by taking the translation component of the inverse of the model-view matrix. This is equivalent to transforming the view origin by this inverse matrix. This computation is performed in the `"updateModelView"` function in `main.js` along with computing the model-view matrix itself, and this function is called after any change to the model or view matrices.

For each triangle of the mesh to be sorted, the distance between its centroid, which is defined in model space, and the model space camera position is computed and stored. The array of triangles is then sorted by these distance values in decreasing order. The now sorted array of triangles is evaluated into an array of indices and uploaded to the appropriate GL buffer, and finally the mesh will be drawn. This method also uses the default shader as in the naïve method, as the difference in operation takes place entirely on the CPU rather than the GPU, except for the final buffering of the updated data.

To facilitate the process of sorting triangles, a mesh contains an array of simple data structures representing the triangles it consists of. Sorting operations primarily operate on this list. The data structure of each triangle stores the following properties:

- **Indices:** a numeric array of length 3, containing the vertex indices that this triangle consists of. By working with indices rather than directly with vertex data, significantly less data needs to be transferred to GPU memory to update the mesh buffers with the new sort order.
- **Centroid:** a 3-dimensional vector representing the model space centroid of this triangle. The centroid is pre-computed as the mean of the position attribute of each vertex, according to the indices described above. This is stored as another numeric array.
- **Distance:** a numeric value storing the most recently computed distance value for this triangle. This is read by a comparison function when the triangle array is sorted.

The sorting process will place the triangle having the largest distance value at the start of the triangle array and that having the smallest value at the end of the array, producing a furthest to nearest ordering. The "SortableBufferedMesh.prototype.sortToCamera" function in helpers.js contains the implementation of this operation. The code of this function is shown in Listing 2, as understanding this operation may be useful later in understanding the novel method.

Listing 2: Triangle sorting (helpers.js)

```
SortableBufferedMesh.prototype.sortToCamera = function(cameraModelSpace) {  
  // Sort by distance to camera  
  // Alternative is to transform centroids to view space and sort by Z  
  // component, but per-triangle matrix multiplications would be slower than  
  // simple distance computation on CPU.  
  
  this.beenSorted = true;  
  
  // Compute each triangle's distance (squared) to the camera in model space  
  let camX = cameraModelSpace[0];  
  let camY = cameraModelSpace[1];  
  let camZ = cameraModelSpace[2];  
  for(let tri of this.triangles) {  
    let dx = tri.centroid[0] - camX;  
    let dy = tri.centroid[1] - camY;  
    let dz = tri.centroid[2] - camZ;  
    // Distance squared is good enough for sorting  
    tri.distance = dx*dx + dy*dy + dz*dz;  
  }  
  
  // Sort by distance, furthest to nearest  
  this.triangles.sort((a, b) => Math.sign(b.distance - a.distance));  
  
  // Update index buffer from sorted triangle objects  
  for(let i = 0; i < this.triangles.length; i++) {  
    let tri = this.triangles[i];  
    this.dataIndexSorted[i*3 + 0] = tri.indices[0];  
    this.dataIndexSorted[i*3 + 1] = tri.indices[1];  
    this.dataIndexSorted[i*3 + 2] = tri.indices[2];  
  }  
  this.gl.bindBuffer(this.gl.ELEMENT_ARRAY_BUFFER, this.bufferIndex);  
  this.gl.bufferData(this.gl.ELEMENT_ARRAY_BUFFER, this.dataIndexSorted,  
    this.gl.DYNAMIC_DRAW);  
}
```

5.3. Weighted Blended Order-Independent Transparency

This is the most conceptually complex method implemented in this application. Unlike any of the other implemented methods, this method involves rendering in multiple passes and formats to multiple render targets. The implementation of this method in the application is a variation on the reference implementations provided in the original report published by McGuire and Bavoil (2013) and a blog post later published by McGuire (2015).

The method is effectively a weighted average blend, where the weighting function approximates the visibility function for each drawn fragment. The weighting function is the default provided in McGuire's 2015 implementation. There exist two intermediate render targets, which in an ordinary implementation are used as an accumulation buffer and a revealage buffer. The accumulation buffer stores a weighted sum of fragment colors, including weighted alpha. The revealage buffer stores the overall revealage for the opaque background. These buffers are written in the first pass, in which geometry is drawn. The second pass is a compositing pass, where the weighted sum of color is divided by the weighted sum of alpha to produce a weighted average, and the inverted revealage becomes the final alpha.

The two render targets require separate blending functions, but the ability to set blending functions per render target is a desktop OpenGL feature which WebGL lacks. Conveniently, as described in the original publication, the buffers can be refactored such that the revealage and weighted sum alpha are exchanged, allowing a common blend function which separates only the alpha function, via `"gl.blendFuncSeparate"`. This variation was implemented in the application.

The implementation was further adjusted to output a correct alpha value, and to output the color in pre-multiplied alpha format for the same reason of specular highlight correctness as mentioned previously. Implementation details for this method are unimportant to this report so no further details are given here. Please see the relevant referenced material for details.

5.4. Novel Method - Multiple Clipped Fixed Ordering

The final implemented method is the novel method developed as part of this project, based on the theoretical design and implementation structure presented in Section 3. This method is named Multiple Clipped Fixed-Order or Multiple Clipped Fixed Ordering, and is abbreviated as "MCFO" when referred to within the application code.

This method consists of two stages. Firstly, all meshes undergo a pre-computing stage when they are first loaded, in which the majority of order computation is done. Secondly, the data produced by this is used in the second stage, which runs every frame in order to render the mesh.

5.4.1. Pre-computation of Fixed Orders

The initial pre-computation stage operates similarly to the triangle sorting method described above. However, sorting is performed along eight fixed vectors and the results are used to create eight dedicated static index buffers containing the required fixed draw orders.

In one of the eight cases, a "distance" value is calculated for each triangle centroid, simply as the sum of the components of its position vector. This is effectively computing a dot product of the position vector with the sorting direction vector (1, 1, 1). Projecting the position onto this vector allows triangles to easily be sorted along the corresponding direction. The resulting sort order will be equivalent to sorting by Manhattan distance to a point along the vector outside the bounds of the mesh. As described in the design stage of the project, this produces a sort order which will be valid in any case that it is clipped appropriately and viewed from the corresponding direction.

Once the triangles have been sorted, their corresponding vertex indices are resolved and then uploaded to an index buffer. However, rather than using a single generic, dynamically updating index buffer like the triangle sorting implementation, one of eight static index buffers is used. These index buffers are themselves numbered in such a way that the buffer number is a bit mask representing the sign of each component of the direction vector used, and the octant it should be clipped to in the next stage.

For the remaining seven cases, as the buffer number changes it acts as a bit mask to selectively make each component of the direction vector negative. In the dot product operation this is equivalent to negating some components of the triangle position vectors before computing their sum. Four of the resulting triangle sort orders will be the exact reverse of the remaining four, however there is no easy way to have OpenGL reverse the order in which it reads a buffer while drawing, so eight full buffers must be used.

The function `"SortableBufferedMesh.prototype.generateFixedSort"` in `helpers.js` implements the pre-computation pass and is given below in Listing 3.

Listing 3: Novel method pre-computation code

```
SortableBufferedMesh.prototype.generateFixedSort = function() {
    // Generate and buffer 8 fixed orders for novel method

    // This function should only be called once after loading or
    // modification, so it does not need to be highly optimized.

    // Order index controls in bitmask form the signs of the vector to sort
    // along, for example order 0 sorts along (1, 1, 1), order 1 is
    // (-1, 1, 1), etc.

    for(let orderIndex = 0; orderIndex < 8; orderIndex++) {
        // Compute vector signs from order index
        let orderXscale = (orderIndex & 1) ? 1 : -1; // bit 0 set negates X
        let orderYscale = (orderIndex & 2) ? 1 : -1; // bit 1 set negates Y
        let orderZscale = (orderIndex & 4) ? 1 : -1; // bit 2 set negates Z

        // Compute dot product of this vector with each triangle's centroid,
        // projecting the centroid onto the vector. This takes the place of
        // distance for sorting.
        for(let tri of this.triangles) {
            let ox = tri.centroid[0] * orderXscale;
            let oy = tri.centroid[1] * orderYscale;
            let oz = tri.centroid[2] * orderZscale;
            tri.distance = ox + oy + oz;
        }

        // Sort by position along vector
        this.triangles.sort((a, b) => Math.sign(b.distance - a.distance));

        // Update index buffer from sorted triangle objects
        for(let i = 0; i < this.triangles.length; i++) {
            let tri = this.triangles[i];
            this.dataIndexFixedArr[orderIndex][i*3 + 0] = tri.indices[0];
            this.dataIndexFixedArr[orderIndex][i*3 + 1] = tri.indices[1];
            this.dataIndexFixedArr[orderIndex][i*3 + 2] = tri.indices[2];
        }
        this.gl.bindBuffer(this.gl.ELEMENT_ARRAY_BUFFER,
            this.bufferIndexFixedArr[orderIndex]);
        this.gl.bufferData(this.gl.ELEMENT_ARRAY_BUFFER,
            this.dataIndexFixedArr[orderIndex], this.gl.STATIC_DRAW);
    }
}
```

5.4.2. Multiple Pass Rendering with Clipping

The second stage of this method occurs in real-time where the meshes are required to be rendered from arbitrary viewpoints. The basic operation of this stage is to draw the mesh in eight passes, using a different fixed index buffer generated in the first stage and different clipping parameters during each pass. This is effectively constructing the full mesh out of eight spatial regions, some of which may be entirely empty due to clipping outside the mesh.

Firstly, before calling an object's mesh to be drawn, the camera's position is projected into that object's model space. This is performed efficiently by inverting the current model-view matrix corresponding to the camera view and object transformation, and then extracting its translation component. This model space camera position is sent to the shader program as a uniform vector.

Next, the vertex position and normal data for the mesh to be rendered are selected and enabled for drawing. Then, a draw pass is made for the eight regions to be rendered. Each of these draw passes begins by determining the parameters for model space clipping and sending this as an additional uniform vector to the shader program. Finally, the corresponding index buffer is selected and enabled, and the mesh is drawn using that index buffer.

The code responsible for this operation is shown in Listing 4, as implemented in the function `"SortableBufferedMesh.prototype.drawFixed"` in `helpers.js`.

Clipping is performed on each fragment within the shader program as a simple component-wise comparison operator between the fragment's model space position, which is interpolated from vertices as a custom varying attribute, against the camera model space position. The comparison determines whether the fragment position is greater than the camera position in each component, producing a Boolean vector result. The components of this result are then conditionally inverted by once again using the region number as a bit mask. Finally, after this, if any of the Boolean comparison results are now true, the fragment is discarded. The result is that only fragments within one of the eight regions are retained. By combining the results of the eight passes of this process, the full mesh will ultimately be constructed, but will consist of the most appropriate of eight fixed draw orders at any given point.

The clipping code is implemented in the `"mcfo"` fragment shader, which is shown in Listing 5. The corresponding vertex shader simply sets the interpolated value `"vModelCoord"` from vertex positions prior to applying any transformation, and computes vertex colors as described in Section 4.7 as is done in other methods' vertex shaders.

This stage ideally would implement, on the CPU, the possible scenarios in which a render pass could be skipped as in Section 3.4. However, this was not implemented in this project.

Listing 4: Novel method multi-pass drawing function

```
SortableBufferedMesh.prototype.drawFixed = function(locationVertex,
    locationNormal, locationOrderVector) {
    // Bind buffers for given locations
    this.gl.bindBuffer(this.gl.ARRAY_BUFFER, this.bufferVertex);
    if(locationVertex >= 0) this.gl.vertexAttribPointer(locationVertex, 3,
        this.gl.FLOAT, false, 0, 0);
    this.gl.bindBuffer(this.gl.ARRAY_BUFFER, this.bufferNormal);
    if(locationNormal >= 0) this.gl.vertexAttribPointer(locationNormal, 3,
        this.gl.FLOAT, false, 0, 0);

    for(let orderIndex = 0; orderIndex < 8; orderIndex++) {
        // Set order vector from bitmask
        this.gl.uniform3f(locationOrderVector, orderIndex & 1,
            (orderIndex >> 1) & 1, (orderIndex >> 2) & 1);

        // Bind selected fixed index buffer and draw
        this.gl.bindBuffer(this.gl.ELEMENT_ARRAY_BUFFER,
            this.bufferIndexFixedArr[orderIndex]);
        this.gl.drawElements(this.gl.TRIANGLES,
            this.dataIndexFixedArr[orderIndex].length, this.gl.UNSIGNED_SHORT,
            0);
    }
}
```

Listing 5: Novel method fragment shader

```
// Uniforms in
uniform vec3 uCameraModelSpace;
uniform vec3 uOrderVector;

// Input from vertex shader
in lowp vec4 vColor;
in mediump vec3 vModelCoord;

layout(location=0) out vec4 target0;
void main(void) {
    // component wise comparison
    bvec3 comparator = greaterThan(vModelCoord, uCameraModelSpace);

    // invert logical value where order vector component is 1
    comparator = notEqual(bvec3(uOrderVector), comparator);

    // clip if any comparisons true
    if(any(comparator)) discard;

    // didn't clip, then normal output
    target0 = vColor;
}
```

6. Results and Evaluation

6.1. Evaluation Procedure for Transparency Methods

The novel method will be evaluated by comparing its performance, visual quality, and applicability with each of the existing methods that were considered and implemented. Evaluation will be performed on a variety of devices including desktop and laptop computers, and smartphones.

Visual quality will be evaluated by identifying visible artifacts, inconsistencies, and any other unexpected output when compared with a conceptual "ground truth" output. The intention will be to provide the most objective evaluation possible, however assessments of visual quality are inherently subjective.

Applicability will be evaluated by reflecting on the functionality of each method. This will also include a reflection on the overall project and whether the developed method has satisfied its requirements.

Using the statistics display described in Section 4.4, performance will be evaluated in terms of the averages of CPU time, GPU time where possible, and the resulting frame rate. Averages will be reset and then allowed to stabilize over a period of one minute per each test. The frame rate will be considered as the primary measure of overall performance, while the per-frame CPU and GPU timings provide a more direct view on how these computational resources are being utilized. In early testing, GPU time measurement was found to be unavailable on smartphone devices.

All testing for the purposes of this evaluation will be performed with identical versions of the application code and using the same settings. The settings used during performance tests will be as follows:

- Camera mode: TEST
- Dark background: (on)
- Load Multiplier: various
- Alpha: 50%
- Algorithm: various

Tests will be performed with every implemented method and will consider the frame rate measurement with load multipliers at values of 1, 10, and 50. This causes the contents of the scene to be redrawn multiple times per frame, simulating larger workloads. This is necessary because the frame rate is limited to 60 frames per second by the web browser, and a small scene at normal load may easily hit this limit preventing accurate measurement of performance. CPU and GPU average time measurements will be taken only with a load multiplier of 1 as time querying has been found to be occasionally unreliable at higher multipliers.

It will not be possible to use a common display resolution between all devices, as the application framework is designed to render at a one-to-one pixel scale and to fill the browser window. However, this should not be a concern, as the intention of the evaluation is not to compare the same method between devices but to compare multiple methods relatively against each other in a variety of environments.

6.2. Evaluation of Method Visual Quality and Applicability

To evaluate visual quality, ideally a ground truth rendered image would be used for direct visual comparison. However, in this case there is no such ground truth image. The image with the highest available quality will be the output from the Triangle Sort method, as the principle of this method is based heavily on the requirements for an exact result. The Triangle Sort output will be taken as the reference image.

The screenshot of the triangle sorting method in Figure 8 shows an overview of the scene from one viewpoint using 50% alpha, and a fully opaque close-up view of an area of complex intersecting geometry where artifacts are visible with all methods.

This method closely represents a ground truth output for the given scene. Most simple geometry appears in the correct order, and alpha values behave as expected. Some complex parts of the scene where geometry is intersecting displays some artifacts. These are most visible where parts of the Utah teapot object model intersect each other, but also appear around the eyes of the Suzanne model. These artifacts appear between two intersecting triangles, where one of the triangles takes full order precedence over the other. This is seen most clearly in the lower half of the figure, where the alpha value is set as fully opaque. Ordinarily a Z-buffer would resolve this problem for opaque geometry, however here it appears as a rough edge arbitrarily following the shape of nearby triangles.

The method is applicable where complex geometry will not be encountered. It may be applied to static or dynamic scenes, although dynamic or more complex scenes will require a large amount of per-frame sorting.

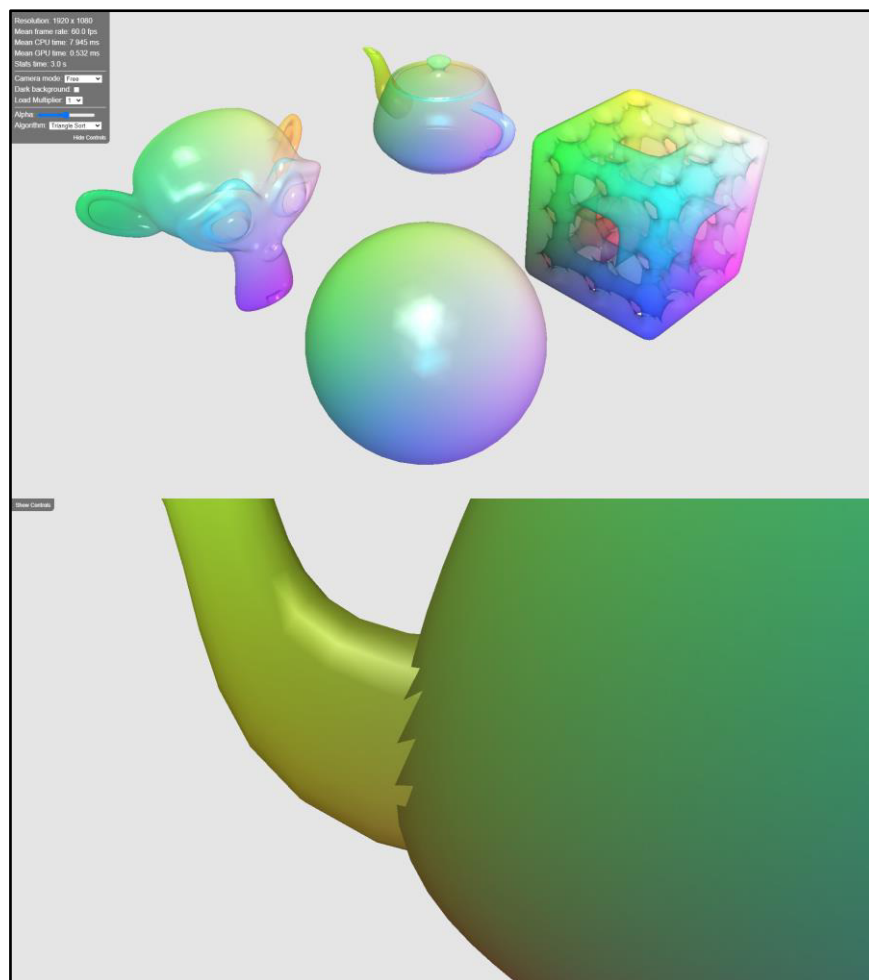


Figure 8: Reference image; Visual evaluation of "Triangle Sort" method.

The following screenshots in Figures 9 to 11 show the same views for the remaining transparency methods.

In Figure 9, we can see that the visual quality of the "No Depth Test" method appears not only incorrect, but highly inconsistent. On the sphere and Suzanne object models, patches of different colors appear where triangles arbitrarily fall in or out of the correct order. On the more visually complex Menger sponge model, it is difficult to resolve any depth information at all due to severe artifacts.

This method is not generally applicable at all for approximating correct transparent rendering. It is considered a worst-case output and represents the exact problem all other methods are designed to solve.

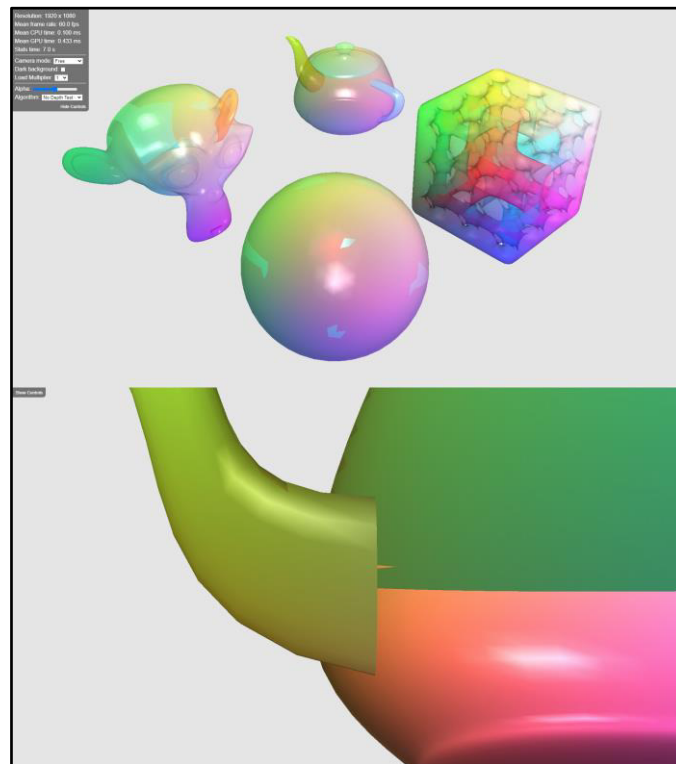


Figure 9: Visual evaluation of "No Depth Test" method.

Figure 10 shows the output of the "Weighted Blend" method. Depth information is generally easy to resolve, and there are no sharp edged or high-contrast artifacts produced. However, there are some noticeable differences between the output colors produced by this method and those in the reference image. This is caused by the way transparency is treated differently in this method, as an input to a weighting function. At lower alpha values, these differences become less apparent. This may be due to the tuning of the selected weighting function. In the lower half of the figure, the region shown is displayed with a maximum alpha value, but the object still appears to be transparent. This problem cannot be solved completely by tuning the weighting function.

This method is highly applicable in scenes where typical alpha values are known in advance, so that the weighting function may be tuned appropriately. It is also applicable in cases where transparency is not expected to be high quality but must retain some depth information. Due to the structure of this method, it may be applied to dynamic scenes just as easily as static scenes.

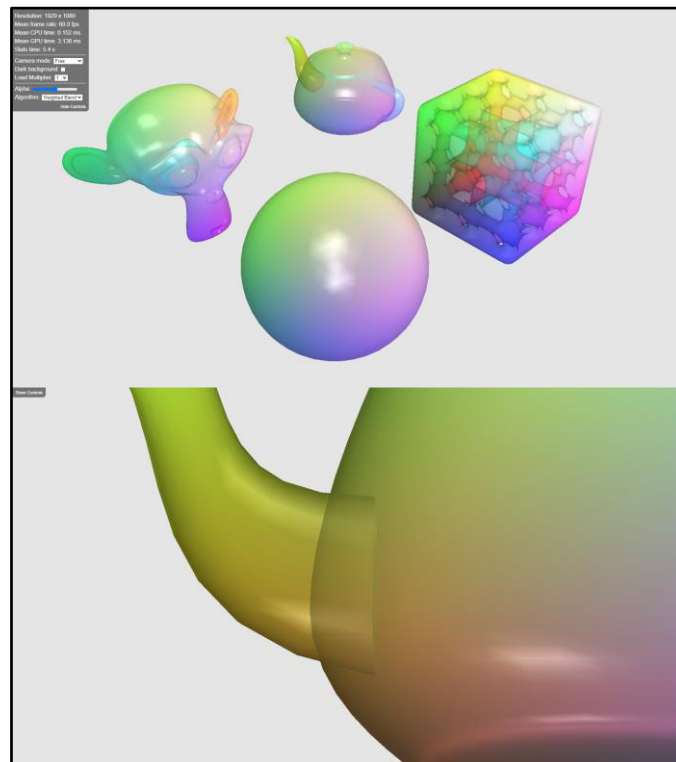


Figure 10: Visual evaluation of "Weighted Blend" method.

Figure 11 shows the visual output of the novel method. The output highly resembles that of the reference image. In fact, the majority of the geometry shown is displayed identically to that of the triangle sorting method. Some artifacts appear in the same situations as that method, although their visual appearance is different. Any method of solving these artifacts which could be employed in triangle sorting, such as BSP trees, may also be employed in the novel method.

This method is applicable only to static meshes due to its design, but it may be used in dynamic scenes if some sorting is performed at a per-object level and object mesh intersections are avoided. The design of this method is optimized for these cases and aims to eliminate the performance impact of per-frame sorting in the more general triangle sorting method.

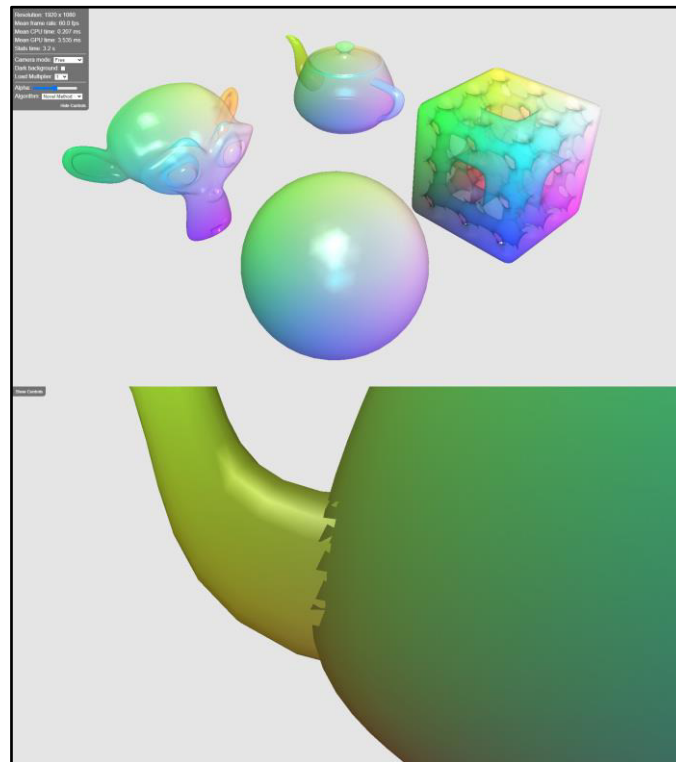


Figure 11: Visual evaluation of the Novel Method.

To summarize, triangle sorting and the novel method produce equivalent, almost correct visual quality while the novel method is more specifically applicable to static meshes. The weighted blending method is even more generally applicable, while achieving a lower visual quality. The no depth testing method is not recommended for use in any case unless the absolute maximum performance is required in favor of visual quality.

6.3. Evaluation of Method Performance on Multiple Systems

6.3.1. High-end Laptop Computer

The first performance test will be performed on a high-end laptop computer. It represents a typical mid to high-end consumer device, expected to run games and other high performance 3D applications such as CAD programs. This system was the primary development environment for the duration of this project, so scene complexity and load multiplier values were chosen to create suitable workloads for evaluation on this system. The basic specifications of this device are:

- Intel 3.5GHz CPU
- NVIDIA GTX 1050 GPU
- Windows 10 operating system with Google Chrome web browser
- Render resolution 1920 x 1080 (full screen)

With the load multiplier at 1 (normal load), all methods achieved the maximum value of 60 frames per second on this device. With the load multiplier at 10, the Triangle Sort method achieved only 37.2 frames per second, while all other methods achieved 60 frames per second.

With the load multiplier at 50, Triangle sort rendered at 9.8 frames per second while the novel method reached 17.4 frames per second, nearly doubling the performance for equivalent visual quality. This comparison is shown in Figure 12. In this test environment, the algorithm has successfully improved the computational efficiency of rendering this workload with near correct results. However, where lower visual quality is acceptable, the weighted blending method is a more efficient solution.

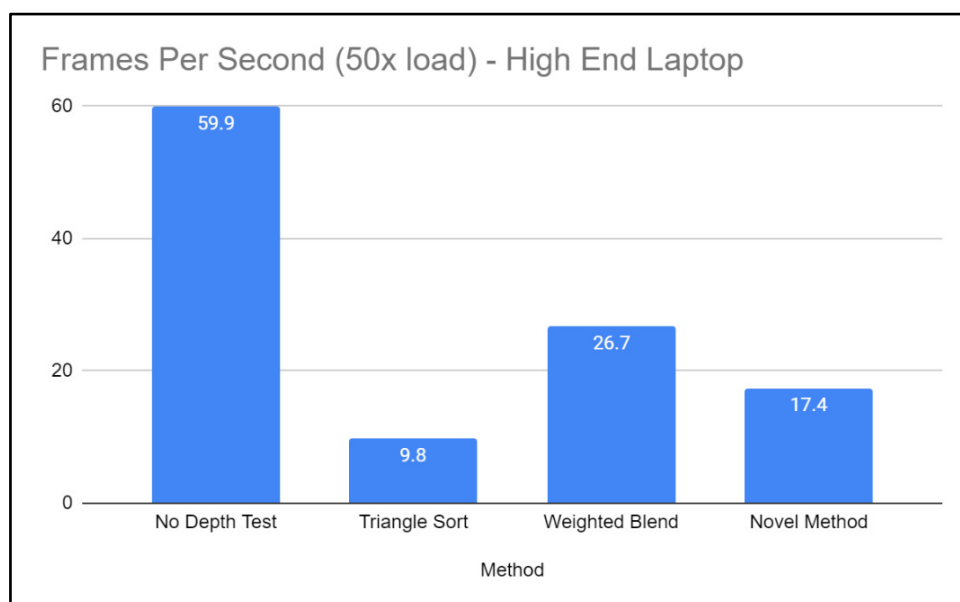


Figure 12: Frame rate test results on high-end laptop.

The computational time will be measured on this high-end laptop system only, as it cannot be measured reliably on all of the tested devices. Timing results are shown in Figure 13, and the results of this test will show how each method utilizes computational resources differently.

It is clear that the Triangle Sort method uses a very significant amount of CPU time compared to any other method, and this is the greatest factor contributing to its frame rate being the lowest of all methods tested. The novel method shows a relatively large amount of GPU time, as well as a small increase in CPU time. The main reason for both of these is the need for eight render passes per object. Variations in GPU time between the other methods is negligible, with the Weighted Blend method being the highest of these, likely as a result of its separate compositing pass.

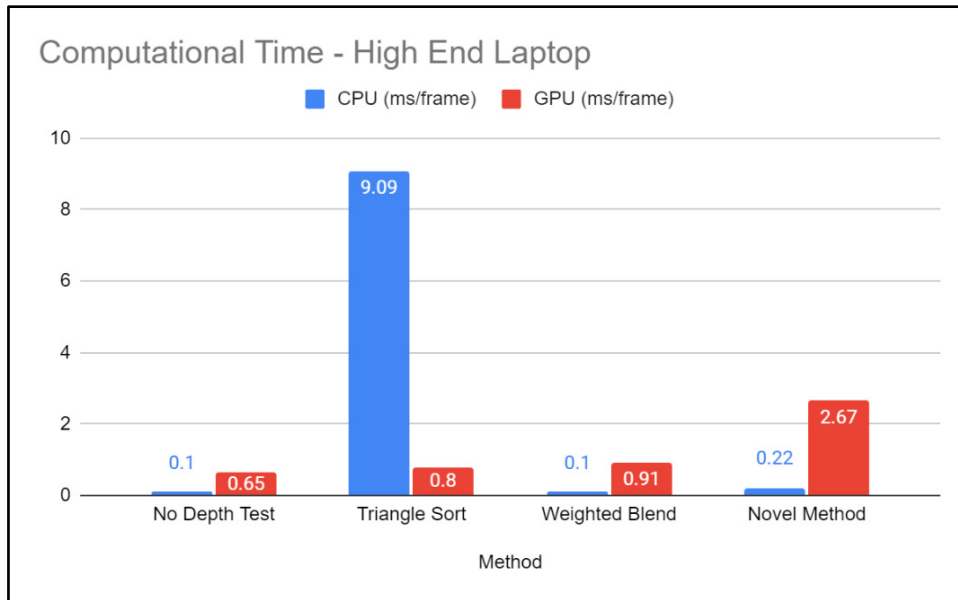


Figure 13: CPU and GPU timing results on high-end laptop.

6.3.2. Performance-limited Laptop Computer

This test is performed on the same system as the previous tests. However, its power management mode has been set to an "eco" mode, forcing both the CPU and the GPU to operate in lower performance modes. The expected result of this change is to show how system resource utilization is affected by the different methods. It will also identify how each method may be affected if the system is to dynamically reduce its performance. This may occur for example if a system under heavy load enters a thermal throttling state to remain reliable.

As Figure 14 shows, when compared with Figure 12, the performance limiting in this test only significantly affected the result of the Triangle Sort method. This indicates that the CPU-intensive tasks it performs are nearing full utilization on this system. The GPU load presented by the novel method for equivalent output remains unaffected, showing that the GPU remains under-utilized despite the heavier workload. However, the simple scene used for testing means that this may not be fully indicative of real-world effects.

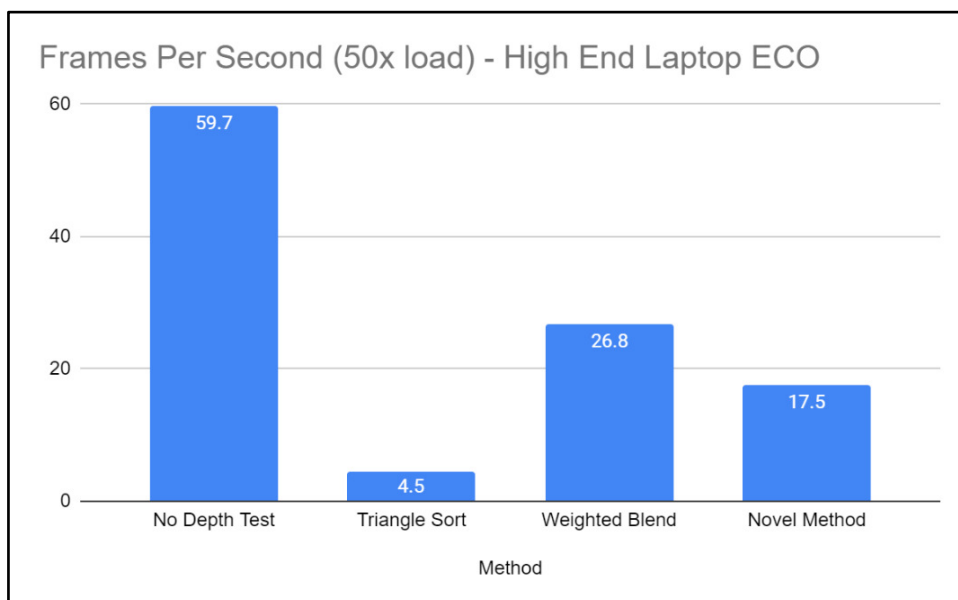


Figure 14: Frame rate test results on performance-limited high-end laptop.

6.3.3. Low-end Laptop Computer

This device is much more limited in all of its computational resources. It represents a lower end consumer device, which may be expected to run simpler graphical applications with lower performance expectations. The specifications are as follows:

- Intel 2.5GHz CPU with integrated Intel HD Graphics 620 GPU
- Ubuntu 19.10 (Linux) operating system with Google Chrome web browser
- Render resolution 1920 x 1080 (full screen)

The results of this test in Figure 15 show that all methods except No Depth Test are limited at approximately the same level of performance. In this case the low-end graphics processing capabilities of the system are limiting the more advanced shader program used in the Weighted Blend method.

It is worth noting that, while this particular test places the novel method and Triangle Sort method at very similar performance measurements, the novel method is not guaranteed to achieve equal or greater performance in general. In a system that is severely limited by the GPU, the method's multiple passes may result in a lower frame rate than that achieved by the CPU-bound triangle sorting.

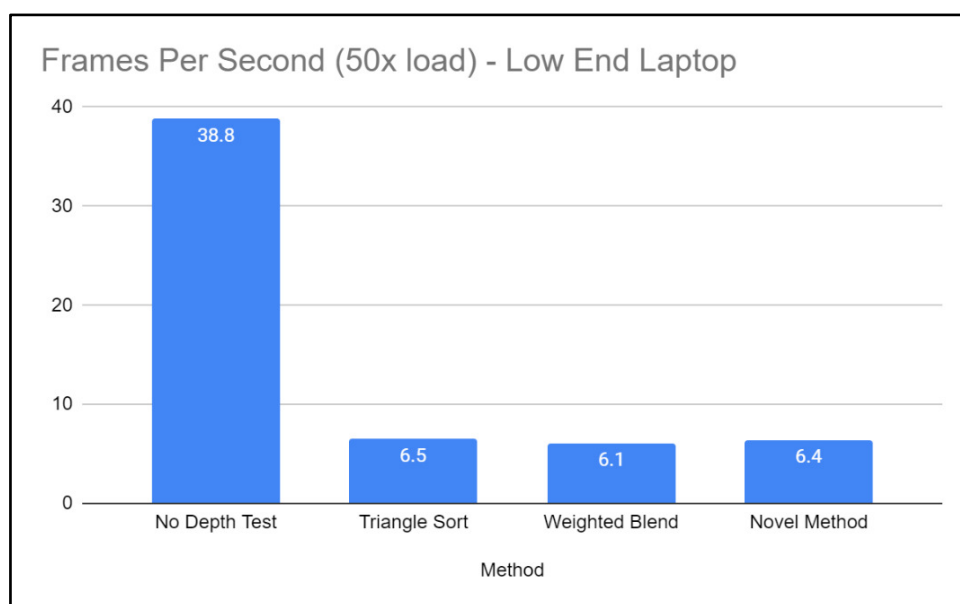


Figure 15: Frame rate test results on low-end laptop.

6.3.4. Smartphone

This test is performed on a mid to high-end smart phone device. This device represents a mobile device that may be expected to run games and simple VR experiences, and other graphical applications occasionally. The performance expectation is low compared to laptop computers. It has the following specifications:

- Snapdragon 660 SOC with 2.2GHz CPU and integrated GPU
- Android 9 operating system (near stock) with Google Chrome browser app
- Render resolution 2080 x 1022 (maximal browser window)

Because mobile devices have lower performance expectations and capabilities, while the test is rendering the same scene at approximately the same resolution as the laptop computer tests, this frame rate test will be performed with a load multiplier of 10 instead of the 50 used previously.

The results of this test in Figure 16 show that the novel method's performance is approximately the same as that of the Triangle Sort method on this mobile device. This may be due to the more limited GPU performance capability of the device, where the higher amount of geometry load presented by the novel method is more likely to be a limiting factor.

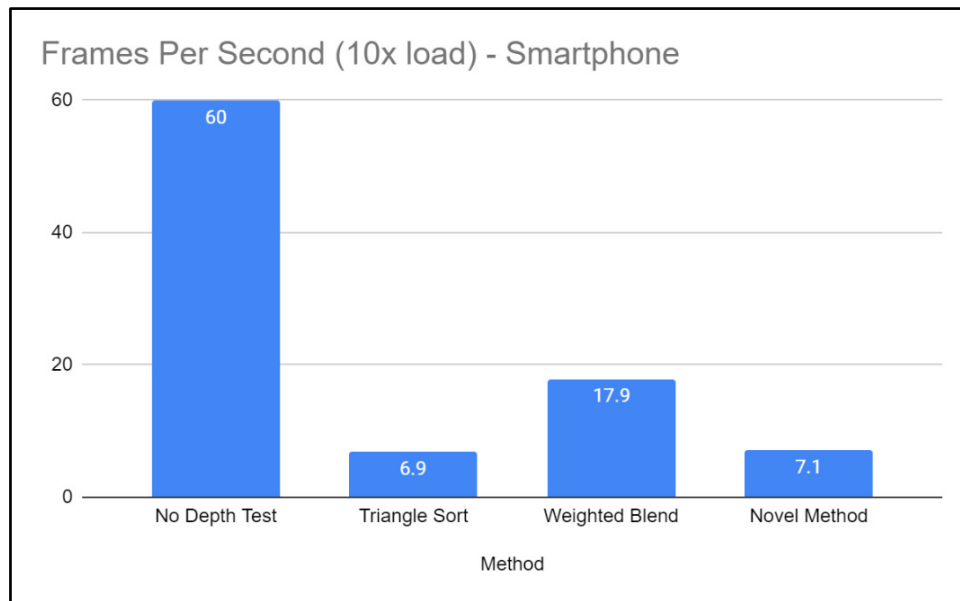


Figure 16: Frame rate test results on smartphone device at 10x load.

6.4. Overall Evaluation

The novel method developed in this project has proven to be generally a more efficient solution to transparency ordering when it is applicable, in comparison with the visually equivalent CPU-bound triangle sorting method. However, the increase in efficiency is not very large due to its reliance on performing eight full render passes. On systems with limited GPU computational power, there is little to no improvement over triangle sorting. The efficiency could be improved in some ways, which are described in Section 7.

Compared with McGuire and Bavoil's Weighted Blended Order Independent Transparency method presented in 2013, their method is able to provide much greater efficiency over either of those more visually correct methods, so it may be favorable in many cases where visual accuracy is less of a concern than performance and consistency.

7. Future Work

Due to time constraints during this project, the developed application did not implement the Dual Depth Peeling method that was initially discussed. For this reason, the novel method could not be compared against this method. This is left as potential future work in any further evaluations of the novel method. The Dual Depth Peeling method also relies on multiple passes, although the number varies with the depth complexity of the scene. A scene with approximately sixteen transparent layers covering a particular output pixel, rendered with this method, would have an equivalent number of passes to the novel method.

As described in Section 3.6, the visual quality produced by the novel method could be improved for problematic areas of complex meshes if a BSP tree algorithm is incorporated into the pre-computation stage. Unlike various existing works investigating the construction and traversal of BSP trees in real-time on the GPU, this would still place the majority of the involved computational load in a "run once" procedure and would not significantly affect the real-time performance of the method. Due to the relative complexity of implementing a BSP tree system, this was not planned to be included in this project and was deliberately left as future work.

Another area for potential improvement is that the implemented method naively renders all eight passes, for which every triangle in the mesh is processed and then clipped at the fragment shader stage. Improvements can be made such that passes which are outside the viewing volume or contain no part of the mesh are skipped. This was mentioned in Section 3.4 at the algorithm design stage, however ultimately the implementation produced for this project did not implement any such culling optimizations.

Finally, although the test case used for evaluation in this project is intended to represent some real-world use cases more accurately than typical test scenes used in previous research, it was artificially constructed and is not truly a test case from real world applications. A more useful evaluation might use real geometry data from a CAD or medical application, or it could simply migrate the novel method implementation to those more standard testing applications and reference scenes.

8. Conclusions

The primary aim of this project was to develop a novel method for rendering static 3D meshes with transparency, with the intention to achieve greater performance than could be achieved with more general methods. The novel method was successfully designed, implemented, and evaluated. The results of the evaluation have shown that the method was indeed somewhat successful at providing an improvement in performance, at least in some test cases. The method is not a perfect solution to the transparency rendering problem, but this was not an expectation of the project. The development process has left the novel transparency method open to future improvements.

The secondary aim was to produce a testing application that would run on multiple platforms in order to demonstrate and thoroughly evaluate the developed novel transparency method and compare it against existing methods. This application was largely a success, as it was able to run on desktop and mobile devices with all implemented transparency methods fully functioning on all platforms. It has provided an insightful set of evaluation results that helped to achieve the primary aim. However, as discussed briefly in Section 7, this application also shows potential for improvement.

Overall, the project has successfully achieved the aims that were initially presented.

9. Reflection on Learning

In addition to this project achieving its practical aims with deliverable results, I feel that I have also gained some personal benefits by completing the project. In the section "Personal Aims" I discussed how this topic of transparency rendering is a particular area of computer graphics research I am deeply interested in, and that I hoped to expand my knowledge in the topic. I feel that the project has succeeded in this regard, as I have had the opportunity to explore in detail some solutions presented by other researchers and ultimately implement these solutions to use them as a part of my own research process.

The project has also been a great benefit to my skills in planning and carrying out a research project from start to finish. By completing the project, I have had the opportunity to research and work on a real computer graphics problem, and this has allowed me to improve the skills needed to do so. I feel that I have been very fortunate to be able to develop these skills by researching a topic that is both important in the current computer graphics field and interesting to me personally.

Appendices

Work Listing by Week

Prior experience:

- Personal experience with some low level WebGL development, including GLSL shaders.
- Understanding of how graphics pipelines use CPU and GPU, and performance implications.
- Awareness of dual depth peeling and other order-independent transparency methods.
- A personal interest in the problem of ordered transparency.

Week 1 (1 Feb - 5 Feb):

- Written the initial plan.
- Initial contact with supervisor.

Week 2 (8 Feb - 12 Feb):

- Discussed the initial plan with supervisor.
- Submitted the initial plan.
- Researched background on the topic, primarily dual depth peeling.

Week 3 (15 Feb - 19 Feb):

- Started developing an interactive WebGL demo application.
- Further background research on WBOIT and other methods.
- Determined the selection of existing methods to implement.

Week 4 (22 Feb - 26 Feb):

- Continued development of the demo application.
- Partially implemented the WBOIT method in the demo.
- Review meeting with supervisor, positive feedback.

Week 5 (1 Mar - 5 Mar):

- Completed implementation of the WBOIT method.
- Implemented the CPU-bound triangle sorting method.

Week 6 (8 Mar - 12 Mar):

- Novel method is partially designed.
- Started writing final report.

Week 7 (15 Mar - 19 Mar):

- Novel method design is completed.
- Continued writing final report.
- Review meeting with supervisor, positive feedback.

Week 8 (22 Mar - 26 Mar):

- Novel method fully implemented in the demo.
- Continued writing final report.

Easter break (27 Mar - 18 Apr):

- Small refinements to the application.
- Continued writing final report.

Week 9 (19 Apr - 23 Apr):

- Preparing demo application for the evaluation stage.
- Bug fixes, optimizations, and code cleanup.
- Continued writing final report.
- Review meeting with supervisor, positive feedback.

Week 10 (26 Apr - 30 Apr):

- Reduced work this week due to CM3202 Emerging Technologies assessed hackathon.
- Bug fixes, optimizations, and code cleanup.
- Continued writing final report.

Week 11 (3 May - 7 May):

- Minor bug fixes.
- Novel method evaluation completed.
- Finishing the final report.

Week 12 (10 May - 14 May):

- Finishing the final report.
- Submitting the report.

Running the Application

No complete code listing is provided in the report. Please see the source code submitted alongside the report.

To run the application, start a local web server from the "app" directory of the provided source code and navigate to it in a modern web browser from any device on the same local network.

For example, using a system with Python 3, run:

```
python -m http.server
```

and point your web browser to localhost:8000 from the same device, or port 8000 on the host device's local IP (192.168.x.x:8000) from another device on the network.

The application's user interface is simple and should be fairly intuitive. Change settings in the menu at the top left corner. Control the camera by clicking and dragging with the mouse and pressing the W,A,S,D keys. Controls vary depending on the "camera mode" setting. See Section 4.2 for more details.

References

- Bavoil, L. and Myers, K. 2008. *Order independent transparency with dual depth peeling*. Available at: http://developer.download.nvidia.com/SDK/10.5/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf [Accessed: 7 February 2021].
- Boman, A. 2020. *webgl-obj-loader source code*. Version 2.0.8. [Source code]. Available at: <https://github.com/frenchtoast747/webgl-obj-loader> [Accessed: 21 February 2021].
- Fuchs, H. et al. 1980. On Visible Surface Generation by A Priori Tree Structures. *SIGGRAPH '80 Proceedings of the 7th annual conference on Computer graphics and interactive techniques*. ACM, New York., pp. 124–133. doi: 10.1145/965105.807481.
- Jones, B. 2020. *glMatrix source code*. Version 3.3.0. [Source code]. Available at: <https://glmatrix.net/> [Accessed: 12 January 2021].
- McGuire, M. 2015. Implementing Weighted, Blended Order-Independent Transparency. *Casual Effects*. Available at: <http://casual-effects.blogspot.com/2015/03/implemented-weighted-blended-order.html> [Accessed: 2 March 2021].
- McGuire, M. and Bavoil, L. 2013. Weighted Blended Order-Independent Transparency. *Journal of Computer Graphics Techniques (JCGT)* 2(2), pp. 122-141. Available at: <http://jcgt.org/published/0002/02/09/> [Accessed: 1 March 2021].
- Yang, J. et al. 2010. Real-time concurrent linked list construction on the GPU. *EGSR'10: Proceedings of the 21st Eurographics conference on Rendering*, pp. 1297-1304. doi: 10.1111/j.1467-8659.2010.01725.x.