Assessment of OpenGl performance techniques

Jahziel Angelo Belmonte

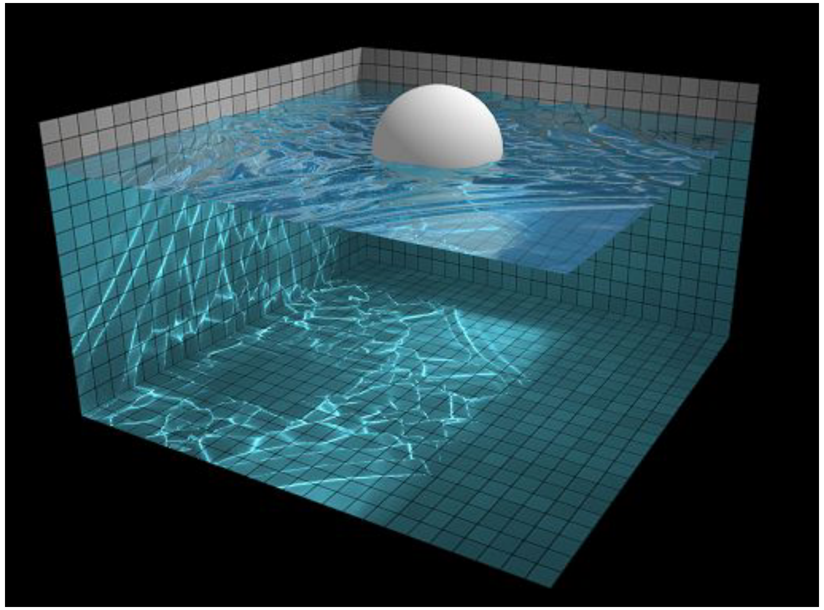
Motivation for picking this project

- Interest in Full-stack development and Computer graphics.

This project will work on WebGL framework which is primarily written in JS (JavaScript) to run OpenGL ES within a web-browser.

Uses web-browsers to render 2D and 3D graphics without the use of plug-ins (Flash player)

WebGL offers interactive front-end experience to users extending front-end stack (HTML, CSS, JS) and deeper dive into computer graphics, specifically the rendering process and how different techniques can improve performance.



WebGL Water
Made by Evan Wallace

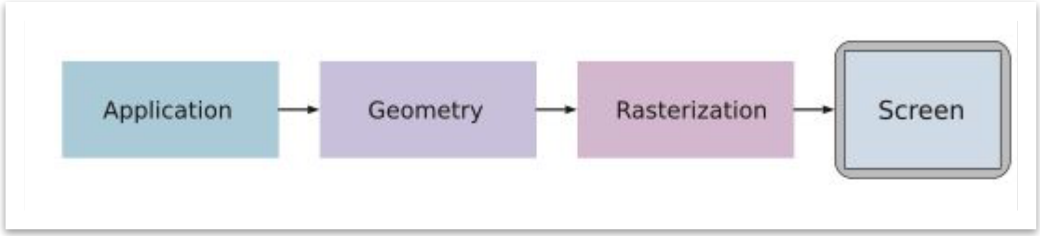
What is WebGL and how does it relate to my project?

- WebGL is essentially a subset of OpenGL. It is a low-level 3D graphics API, at the most basic level most of the differences are in the programming language constructs C++ vs JS.

- OpenGL allows the programmer to interact with the GPU (Graphical processing unit) to achieve hardware accelerated rendering

- When drawing an image on your monitor, the CPU and GPU are both involved in steps called Graphics pipeline. This is necessary for transforming a 3D scene into a 2D representation on a screen.

- This task can become heavy on the GPU quickly when drawing many polygons/triangles (some not visible to the camera)



Graphics pipeline: Wikipedia

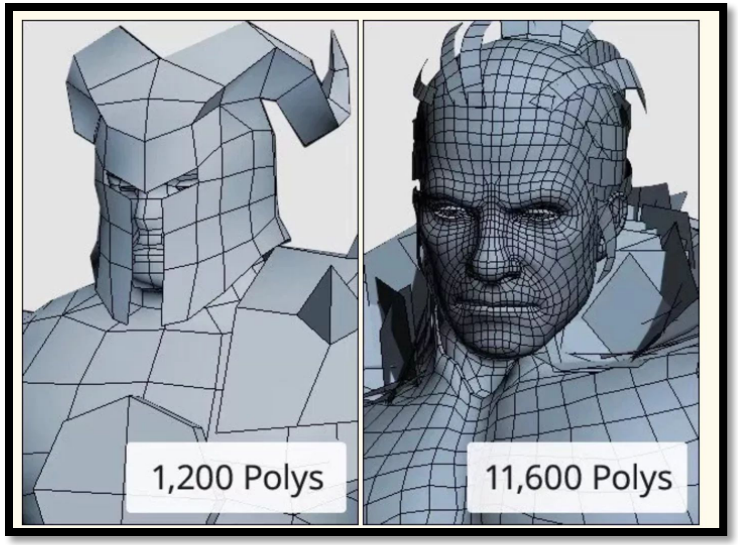
When would we need optimization ?

- High Polygon count (Complex Geometry), The GPU has to process each polygon so for large number of N polygons will increase calculation and render time.

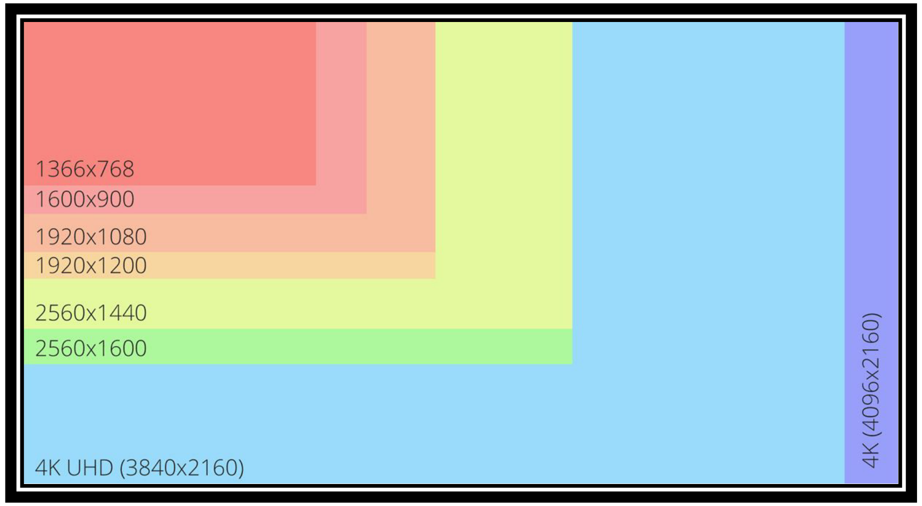
- Complex Shaders (Advanced Lighting & Effects), Simple shaders might just apply color, but complex shaders include realistic lighting, or calculate light interaction with surfaces

- High Resolution (Textures and Screen Size), At High resolutions (4k) the GPU may need to process over 8 million pixels per frame. When running a process at 60 frames per second the computational size becomes exponential.

- Fill Rate Overload (Fragment Processing), Fill Rate refers to how quickly the GPU can fill fragments (Pixels) on the screen. This can become a bottleneck within the GPUs workload.



What is Polygon count: Delasign



Information about Screen Resolution: Logical Increments

Current Optimization techniques

- Degenerate Triangles

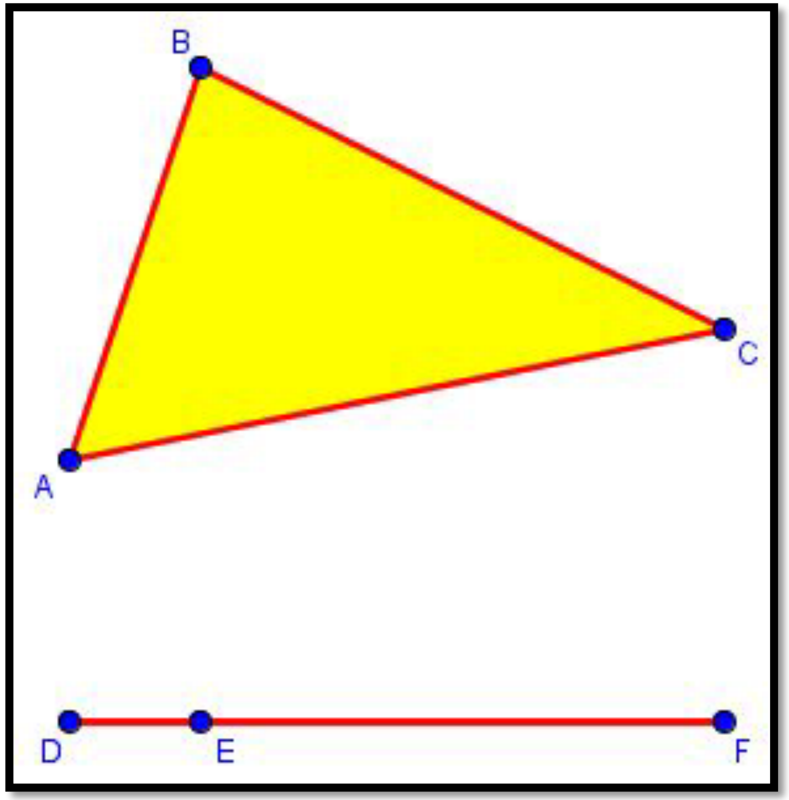
- This is a triangle that has no area as two or all vertices have the same coordinates (co-located)

- Often used in triangle strips (Triangle mesh) to create a continuous sequence of triangles while avoiding gaps.

- These can reduce the amount of draw calls which is more efficient for the GPU. This slightly improves performance each call involves communication between the CPU and GPU.

- Trade-Offs:

1. Some performance impact, GPU will still need to process these triangles causing a small overhead, modern GPUs can handle these cases efficiently



Degenerate Polygons: The Math Doctors

Current Optimization techniques

- Interleaving (Interleaved Vertex Data)

- The practice of organising vertex data so that attributes such as (position, colour, texture coordinates) are stored together in memory rather in separate arrays

- Modern GPUs fetch memory in chunks. When interleaving vertex data, this data can be gathered in one fetch operation.

- This can speed up memory locality and cache performance, reducing memory accesses and rendering speeds.

- Trade Offs:

1. Cache Miss: If some attributes like colours, texture-coordinates are not needed it results in cache inefficiencies. GPU may fetch more data than needed resulting in wasted memory bandwidth.

```
[posx,posy,posz,color1,color2,color3]
[posx,posy,posz,color1,color2,color3]
```

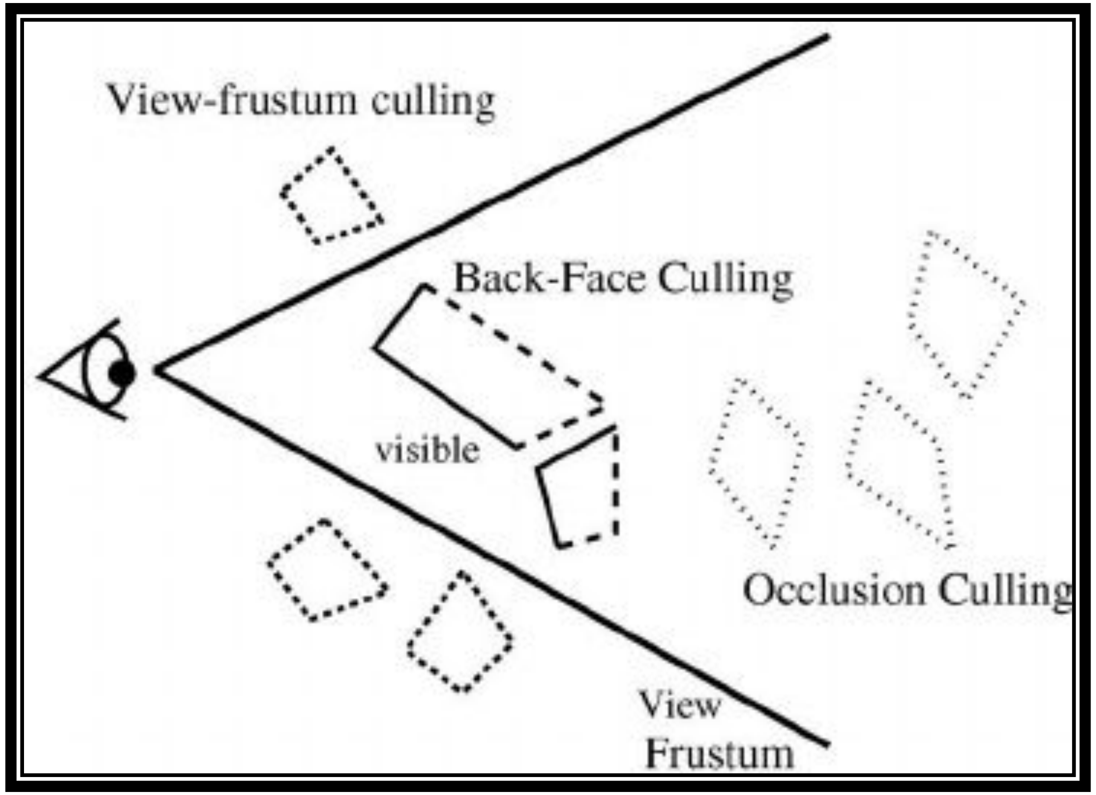
Ex: Storing Position and Color data
By using an interleaving format, data can be fetched in one operation

Current Optimization techniques

- Frustum Culling

- Determines which objects in the scene are within view and only rendering said objects.
- If the GPU were to render all objects including non-visible ones, this would be wasteful in memory bandwidth and GPU efficiency
- Culling these objects can significantly reduce amount of draw calls and GPU processing time.
- Trade Offs:

1. CPU Overhead: Since this processing is done before the GPU the CPU has the responsibility with this task.
2. Quality impact: If the culling is too aggressive, objects may disappear too early resulting in (popping or clipping artifacts)



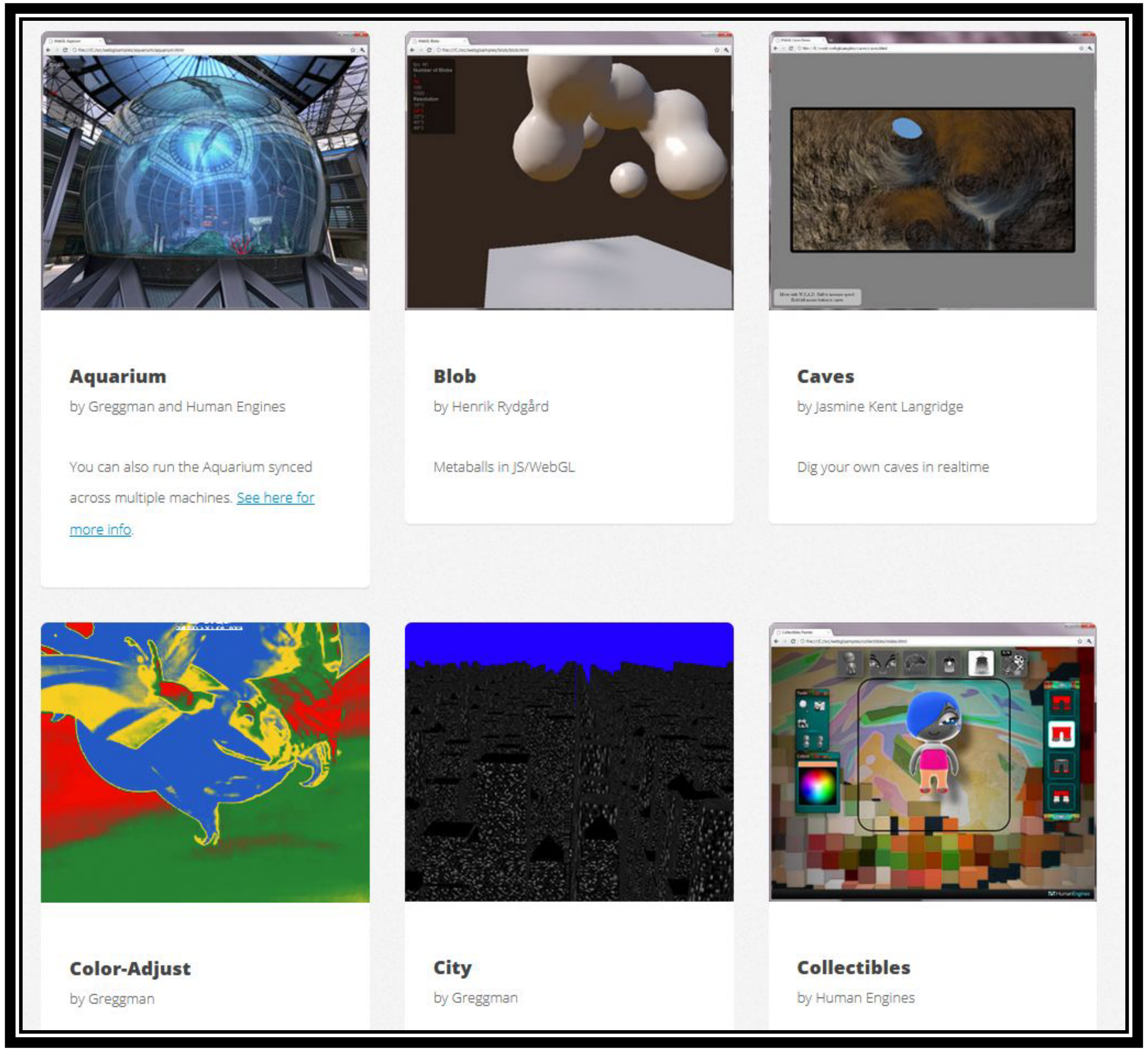
CPU/GPU Camera culling – [Blender Community]

Relation to Machine Learning

- Dynamic Optimization:
  - Training a model to adjust rendering techniques (Texture quality) based on current GPU, CPU metrics. The model could learn which graphics settings yield best for the user's system.
- Algorithm Selection
  - Determine which optimization technique to use for specific scenarios based on the current user camera position and scene.
- Resource Allocation and Predictive Caching
  - Optimizing resource allocation for rendering tasks. Using previous usage logs, the model can recommend how to allocate/cache memory. Prediction models could determine which model/texture will be used, the system could pre-load these reducing fetching times.

Goals for the project

- WebGL demo scenes comparing normal rendering techniques against assessed rendering techniques.
- This would involve comparing rendering quality, frame rate, Hardware compatibility, CPU and GPU load.
- Using ML, demonstrate Real-Time adaptation which selects the best rendering techniques based on the scene and user's device.
- Future integration could involve WebVR or Vulkan (Cross-platforming)
- Integration with Node.js (Gaas)
  - Server-Side rendering – Run headless rendering within the server and send the output to the client.



WebGL Samples