# CSci 5608 Spring 2018

**Written Assignment** **Xingzhe Xin, xinxx073@umn.edu, 5197296**

## LEVEL OF DETAIL ALGORITHMS

Games have always fascinated me, and the graphics in games have become closer to reality than ever before. Most recently, Ureal Engine has been able to capture a person's whole body and facial expressions and render 3D animation with the same expressions in real time.[1] As impressive as this may be, the video was rendered in real time by 4 GTX TITAN V graphics cards, each costing around $3000. Before TITAN V level computing power arrives in everyone's ultrabooks, we must make use of the limited hardware resouces we have.

Level of detail algorithms exist for this purpose. If we are watching a movie at a theater, the framerate is usually a fixed 24 fps in order to produce a stable, steady experience to the audience. However, we soon discover that 24 fps does not work well in games. When a movie is being filmed, the camera is either not moving at all, or moving at a very steady rate while pointing at some fixed position. However, within a game, the main camera is constantly changing direction – especially in First Person Shooters, where the camera is constantly pointing left and right aiming for the enemies. Assuming that the monitor can display images at high frame rates, he gaming experience increases dramatically when increasing the fps from 30 to 60, 120, and finally 240 fps, above which, the increase is close to unnoticeable. With this discovery, games have been tring to maximize the amount of fps by lowering the level of detail each image produces, and this is when LOD algorithms comes in handy.

The least suitable technique in this case might be image-driven-simplification due to its slow nature.

With vertex clustering, we could keep the total number of verticies in the scene below a certain amount. Since when the number of verticies is stable, the amount of triangles in the scene might also be close to some constant value. With this method, video memory useage should also be relatively stable, since it stores the texture and vertex information of the scene. For different machines, we just need to change this "optimal reference number", which is basically just the number of cells, to adapt to the machine. The number could simply be calculated with a hardware benchmark program that executes when the game runs for the very first time. Vertex clustering also seems to be easy to implement. Comparing this to floating-cell clustering, vertex clustering should operate faster, since it does not need to sort the verticies by their importance. Although it might not look as good as using floating-cell clustering, games are, more often than not, performance driven, and vertex clustering might be the better option if we want to keep the frame rate steady. This algorithm is simplistic at its core, so it can easily be used in different situations, across platforms that have varying performance.

[1] Andy Serkis - Unreal Engine `https://www.youtube.com/watch?v=TxErDzsIdKI`

Vertex decimation better preserves the shape of the object by keeping track of the topology, but it is hard to guarantee an outcome. When we specify the distance to edge and distance to plane, we can't directly see how much impact we are making to the outcome. Those criteria are also difference when the scene changes, thus the implementation should be harder. To stay at a constant frame rate, the parameters passed in to the algorithm would need constant adjustment.

With the two clustering techniques, we could control the average triangle size by limiting the size of the cell we are using. This seems quite a difficult task for vertex decimation, since the poly count after decimation is hard to determine.

And finally, the quadric error metric algorithm stands out due to its preservation of the topology while being more "controllable" than the vertex decimation technique.

---

a. Object Coherence
   We make use of the fact that most object surfaces are locally consistent. So when we render one pixel on screen that has the color of an object, the next pixel is likely to have roughly the same color as the current pixel. This could be used in resolution scaling in games, for example when we set the "render scale" to 25%, for every 4 pixels on the screen, we get the value from one pixel and duplicate that value to 3 other pixels.

b. Face Coherence

c. Edge Coherence
   – Image based rendering –

d. Scan-line Coherence
   When moving from one scan-line to another, the changes become predictable, and the intersection between the edges become easy to compute.

e. Area Coherence
   warnock algorithm

f. Depth Coherence
   binary space tree

g. Frame Coherence
   Frame coherence basically means that we could use the information from the previously rendered frame because of the fact that in many cases, there aren't drastic differences between two consecutive frames (i.e., they should look rather similar most of the time). So we could use this characteristic to our advantage. One example would be data compression for videos. If, for example, it's only the main character doing something while the background doesn't move at all, there is really no need to store the full

frame, since we could just use the same color as the last frame on those areas. Another application that I've seen in real life is that on some modern projectors, it is able to take a 24 fps movie as input but project at 60 fps. This is the exact same idea of utilizing the similarities between frames and construct filler frames that make the animation look much smoother. It does have its issues - when the camera rapidly changes position or just the scene is changing from white to black quickly, there is visible ghosting and sometimes causes discomfort for the viewers.