COMP4610/COMP6461

Week 9 - Visible Surface Algorithms, Clipping and Shadows

<Print version>

Admin

Assignment 2

- Assignment 2 has now been released.
- If you would like to self-select a team, register your team by **5pm tonight**

  `https://anu.au1.qualtrics.com/jfe/form/SV_9ZgePqjQdtOiCLc`
- Unregistered students will be automatically allocated to teams, and notified on Monday.

Visible Surface Algorithms

Visibile-Surface Detection

- For displaying 3D scenes it is useful to be able to determine which surfaces are visible.
- These algorithms are referred to as **visible-surface detection** or alternatively **hidden-surface elimination**.
- Methods can either be in:
  - object space - works at the object level.
  - image space - works at the pixel level.
- In most cases algorithms form a **potentially visible set**, which includes all visible elements, and, perhaps, some non-visible ones.

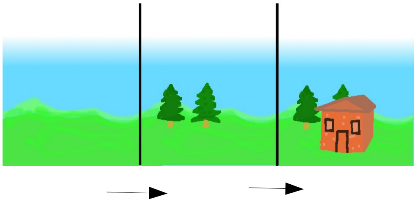
Potentially Visible Set

- We want an algorithm that outputs a list of surfaces, which includes **all** visible surfaces.
- Outputting everything would work, but it is (probably) not efficient.
- The key is to *quickly* remove objects that are clearly not visible. If a few non-visible objects are still included, this is fine.
- Example: Let I be an objects bounding box intersects the camera's frustum. And V be an object being visible on camera. We have

$$\neg I \implies \neg V \text{, however}$$
$$I \not\Longrightarrow V$$

The Painter's Algorithm

- A simple way of drawing a 3D scene is to order the objects by the distance from the viewer and paint them from furthest to closest.
- This an image space solution, and simple and works well.
- When using transparency this is often the only solution (efficient order independent transparency remains an unsolved problem in computer graphics).
- Sorting objects can be slow. Ideally we want a sorting algorithm that is quick when there are few inversions **[why?]**.



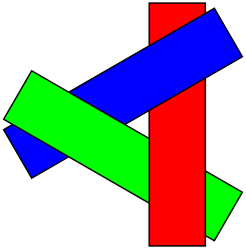
Visible Surface Algorithms

Image Based

The Painter's Algorithm

Disadvantages

- Overlapping problem.
- Sorting triangles is slow.
- Overdraw.



What order makes this object draw correctly?

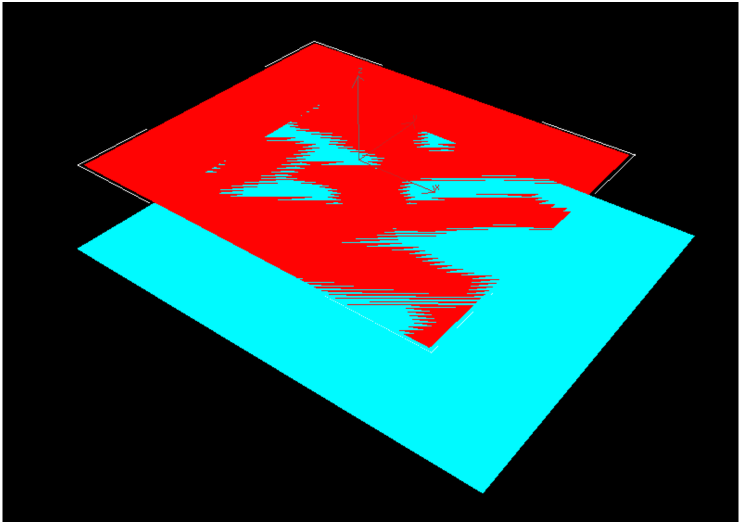
Depth Buffers

- One criticism of the painters algorithm is that you may end up redrawing pixels multiple times. This is called **overdraw**.
- Reverse painter's algorithm draws objects in reverse order (closest first), while keeping track of the z-depth for each pixel.
- Just before a fragment's color is calculated a quick z-depth test is performed. If the new pixel is further away than the previous one it is discarded, and no calculations are done.
- Depth testing can be enabled in OpenGL via **GL_DEPTH_TEST**, but remember, if you use this to also clear the depth buffer each frame using **GL_DEPTH_BUFFER_BIT**.
- The depth buffer allows any order of rendering, therefore implementations of reverse painters algorithms almost always sort **objects**, rather than **individual triangles**.
- The depth buffer also solves the overlapping problem, and is therefore almost always enabled.

Z-Depth fighting

- Typically, we store z-buffer as a 32bit float (although some [older] engines may use 16bit float).
- Normalized coordinates are used, which means your ‘near’ plane is at -1, and your ‘far’ plane is at 1. Therefore if you set the far plane to very far away, you may lose precision on your z-buffer.
- When two triangles exist on the same plane, they will have mathematically the same z-depth. However, small rounding errors will occasionally mean that their depth values will differ. This causes ’z-fighting.
- The solution is not to have two objects overlapping on the same plane. I.e. if you add a poster to a wall, offset the poster by a slight bias.

Z-Fighting



Two co-planar quads.

Advanced: Differed Renderer

- One way around the problem is to take advantage of the fact that coloring the fragments is the slowest part. Therefore we can simply 'draw' the scene by writing everything needed to a **g-buffer**. This would include the 3d position, the normals, as well as albedo and specular.
- We then process the lighting in a single pass using the g-buffer as input. This way we guarantee that lighting is computed only once per pixel, making our worst case scenario much faster.
- Differed renders have many limitations (how to handle transparency for example), but are quite common.
- There's an interesting discussion about the choices between differed and forward rendering engines here: https://blog.theknightsofunity.com/forward-vs-deferred-rendering-paths/.

Visible Surface Algorithms

Object Based

Back-face Culling

- If a triangle faces away from the camera, then do not draw it.
- This is very easy to check this (clockwise / anticlockwise)
- For a triangle, this is simply checking if the normal's z points into the screen or out.
- These normals are usually calculated on the fly from the 3 points of the triangle.
- This will remove about half the triangles.
- In OpenGL, we get control over how things are drawn on the 'front' and 'back' face. e.g. we can set lighting for the front face to be different from the back face.

Face Culling in OpenGL

OpenGL allows the user to specify how to cull faces.

```
glEnable(gl.GL_CULL_FACE); // enable culling
glCullFace(gl.GL_BACK); // decide if we want front or back faces to be removed
glFrontFace(gl.GL_CW); // define the order for the front face.
```

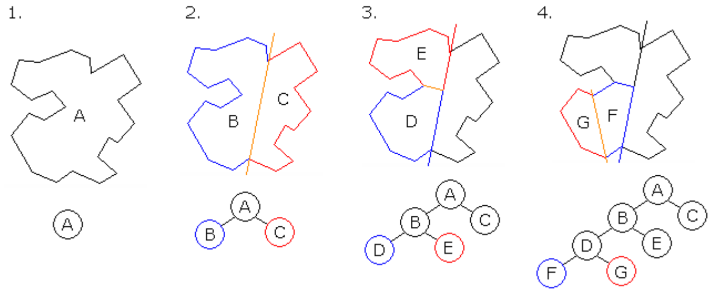
Frustum Culling

- Very simple idea. Check if an objects bounding box intersects the viewing frustum.
- Some care is needed if we are using shadows (as off-screen objects affect the lighting of onscreen objects).
- Organising objects into a grid can make this very efficient.
- Minecraft and others do this this with chunks. However, in this case you would need to load nearby chunks and update them, but only draw the ones infront of the camera.

Binary Space Partitioning (BSP)

- This is an advanced method of quickly identifying a PVS for **static** objects.
- First used in the Doom engine.
- How you divide up the scene matters.
- Must cut objects when the division plane intersects them.
- Generally not used much anymore, but was very popular (and important) for a long time.
- Developed in 1980 (`http://www.cs.unc.edu/~fuchs/publications/VisSurfaceGeneration80.pdf`) but was not commonly used until 13-years later.

BSP - Example

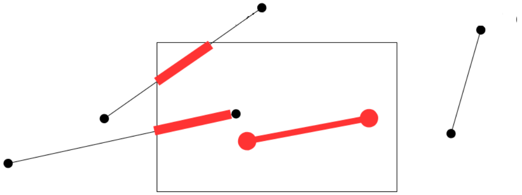


For rules for traversal see https://en.wikipedia.org/wiki/Binary_space_partitioning

Clipping

Clipping

Sometimes objects you are drawing sit outside the drawing area. These should be clipped to the drawing area. This could be done in image space. However, it is generally more efficient to do it in object space. Objects completely outside the clipping area can be culled. They can be left unchanged if they are entirely within the clipping area, whereas those that border the clipping area need to be modified.



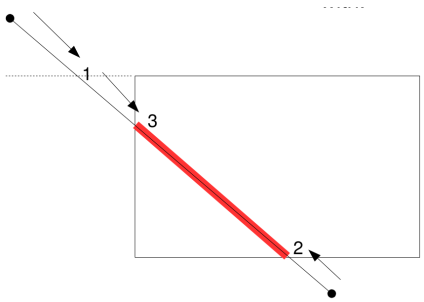
Below is an example of lines being clipped to a rectangular area.

Cohen-Southerland Line Clipping

- The Cohen-Southerland line clipping approach is a very old approach that aims to efficiently deal with lines that are either completely within or completely outside the clipping area.
- This approach works be creating region codes for the end points of the line. A region code is made up of 4 bits corresponding to TOP, BOTTOM, LEFT, and RIGHT. So if the point is above the rectangle then the TOP bit is set to 1 otherwise it is set to zero, if the point is to the LEFT of the rectangle the LEFT bit is set to 1 otherwise it is set to 0, etc...
- The two region codes for the points can be bitwise or(ed) and if the resulting code is 0000 then the line is completely within the clipping area and can be left unmodified.
- The two region codes for the points can be bitwise and(ed) and if the resulting code is not 0000 then the line is completely outside the clipping region and can be culled.

Cohen-Southerland Line Clipping

- In other cases the line can be intersected with a boarder of the rectangle reducing the length of the line. This processes is repeated until either line is removed completely or it sits within the clipping rectangle. Note that the region codes can be used to determine which board to interest the line with.



Example of Cohen-Southerland Line Clipping

Shadows

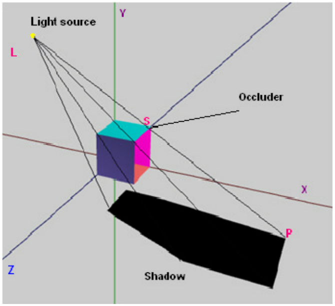
Overview

Shadows are a bit tricky (you have to consider **all** other objects when color for **each** pixel...) $O(N^2)$. Some common methods are

- **Raytracing**: easy, but slow.
- **Shadow Volumes**: not used much anymore, gives very sharp shadows.
- **Shadow Mapping**: a good compromise but has problems.
- **Pregenerate** the lighting, save to a light map (Unity and others have this), then have a second system for dynamic lights/objects.

Projection Shadows

- Project a 3D object onto a 2D surface, then draw it using a dark colour.
- Instead of projecting an object, you can project a sprite, often a circle.

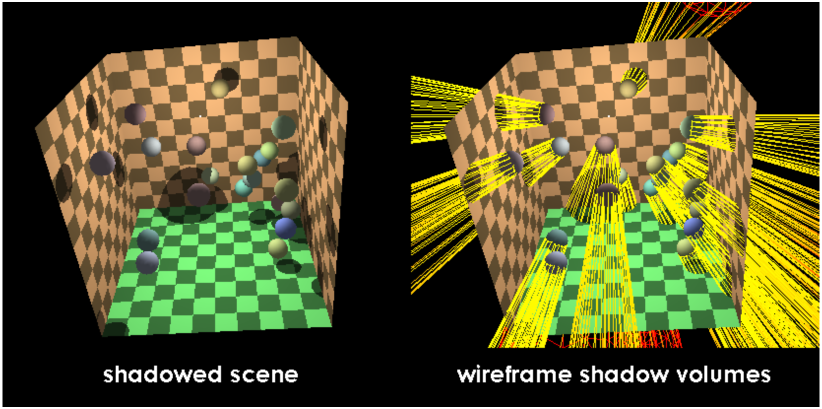


Credit https://www.researchgate.net/publication/236592293_Real-Time_Shadow_Using_a_Combination_of_Stencil_and_the_Z-Buffer (CCA 4.0) international

Shadow Volumes

- Produces very sharp shadows using the stencil buffer.
- Shadows look very sharp.
- Doom3 uses this.
- Requires a Stencil buffer.
- Includes an algorithm called the "Carmack reverse" which John Carmack invented, but is not allowed to distribute, as it was somehow patented by another party.

Shadow Volumes



Credit https://en.wikipedia.org/wiki/Shadow_volume (CC BY-SA 3.0)

Shadow Maps

This is very common, 9 times out of 10, shadow maps are what you want to be using. It works by...

- Calculating a depth map based on the distance from the light's position (in a similar way to calculating the texture in the previous slide).
- Then as each fragment in the scene is rendered calculate the distance between that fragment and the light source.
- If this distance is larger than the depth one looks up in the depth map then the fragment is in shadow.

This is quite fast - requires just 1 texture lookup, but does require rendering the depth map from the perspective of each light. Might be best to make some lights not cast shadows, or some objects to not cast shadows, or use simplified models for shadow casters.