Introduction to Graphics Programming

This tutorial will introduce you to the concepts behind modern graphics programming, including how vertex data is turned into a fnal pixelated image. You'll be introduced to the APIs that can be used to render graphics, the types of data used during rendering, and how graphics hardware is programmed.

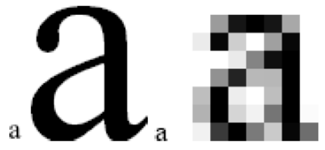
Rendering Techniques

Ray Tracing

Ray tracing is a popular rendering technique that can generate very realistic images, by simulating how light works. As its name suggests, ray tracing works by casting rays through a scene, and seeing which objects it intersects; depending on the material properties of the object, further rays may be created to find objects that may cast reflections on an object. As the rays propagate through the scene, they 'pick up' the colours of the objects they bounce off. Contrary to expectation, ray tracing generally works by casting a ray into the scene for each pixel of the screen, rather than casting rays from each light in a scene. While this technique can create realistic images, it is very slow, due to the large number of ray calculations required for each frame. Despite this, ray tracing is often used in the film industry to render CGI scenes as visual fidelity is of greater importance than speed, making rendering times in the region of 1 frame every 7 hours (as in Toy Story 3) a worthwhile tradeoff.

Vectors

Vector graphics are made up of a number of mathematical functions, defining lines, points, and curves that create a final image. This mathematical property allows a vector image to have 'unlimited' resolution when zoomed into or made larger, unlike a bitmap image which will get blocky as its pixels take up more screen space. A classic example of vector graphics are fonts - one 'set' of functions defines a font that can be scaled to any point size with no loss in detail. Macromedia Flash is another popular technology that can utilise vectors, both for text and animated graphics.



Example of Vector graphics being smoothly rescaled. Left: Vector Graphic Right: Bitmap

Rasterisation

Generally speaking, rasterisation is the process of taking image data, and displaying it on screen; it so called because of the scanning pattern of old Cathode Ray Tube monitors, where an electron beam 'rakes' (latin: rastrus) across the screen in a gridlike pattern. More commonly though, rasterisation refers to the process of projecting a series of geometric shapes, usually triangles, onto the screen. Unlike ray tracing, rasterisation doesn't calculate the light and shade of a pixel via object intersection, and usually does not use curve calculations as vectors do, making rasterisation very fast. The pixels determined to be covered by a geometric shape can be shaded though - meaning its colour can be calculated from information relating to the covering shape, such as its texture and facing direction. Due to this speed and adaptability, rasterisation has become the most popular method of rendering 3D scenes in computer games, and several rasterisation APIs have been developed to aid in graphical rendering using rasterisation.

Graphical Data

In modern graphical rendering, there are three basic forms of graphical data, vertices, textures, and shader programs. When in use, this data is generally contained in the onboard RAM of your graphics hardware, but can be loaded in on-demand, or cached in system memory if onboard graphics memory is insufficient.

Vertices

A vertex (plural *vertices*) represents a corner in a shape. All of the shapes used in game rendering are made up of a number of flat surfaces, with vertices as edges - even 'round' shapes like spheres! Vertices are connected together via edges, forming primitives, which can then be coloured or texture mapped accordingly. Vertices have at least a position, generally defined in $\mathbb{R}^3$ Cartesian space, but may have a number of additional *attributes*: colours, texture coordinates, normals, and other information required for advanced rendering techniques.

Textures

Textures are the images applied to the geometric data you want to render on screen. They are generally 2D, loaded from an image file format like PNG or JPG; 3D textures are sometimes used for certain types of advanced visualisation, such as the results of a medical MRI scan. 1D textures are also possible, think of them as a single strip of pixels - or rather, *texels*, the proper name for each component of a texture. Each of these components has a number of values - usually this is *red*, *green*, and *blue* information, sometimes with an additional *alpha* channel to denote transparency. Some hardware APIs (OpenGL 2 does, OpenGL 3 does *not*) support 'palletised' textures, where the texels in a texture are indices into a colour palette - this was a popular method of defining graphic information in the 8 and 16bit console era, but is rarely used now.

Shader Programs

Shaders are short programs that run directly on your graphics hardware, and which perform the operations that turn your graphical data into the desired final image. Like the C programs you have been writing, shaders are written in a high level language, and then compiled into a binary shader executable, suitable for running on your graphics hardware. Shader programs can be split up into three different types: Vertex, Geometry, and Fragment. Each shader *program* has a different scope, and takes in, and outputs, different types of data - these different shader programs can be combined to form a single shader *executable*.

Vertex shader programs, as their name implies, run at the vertex scope - each execution of a vertex shader can 'see' a single vertex, which it transforms into the desired position on screen, and calculates any additional data that the further shader stages may require. Vertex shaders output transformed vertices.

Geometry shaders operate at the primitive stage - they 'see' a complete primitive (such as a line, or a triangle), and output 0 or more primitives. Geometry shaders are often used to 'amplify' incoming geometry to a higher level of data, possibly based on distance to the camera, in a process known as *level of detail*. The input and output primitives don't need to be of the same type - a geometry shader can turn lines into triangles, or vice versa.

The final shader type is the **fragment**. As clipped triangles are rasterised, the potential pixels, or fragments, that they cover on screen can be determined, and sent the the fragment shader. This has a scope of a single fragment, and takes in interpolated vertex data, and outputs a single value - the colour of the fragment, which may be any combination of texture data and vertex attribute.

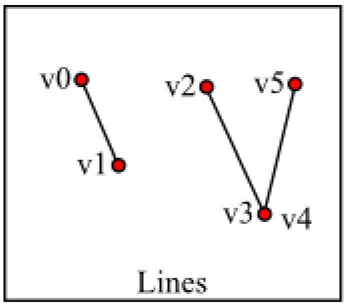
Modern graphics hardware has many cores, running into the hundreds, or even in the case of multi-chip cards, over a thousand - each of these cores can run either a pixel, vertex, or fragment shader at any one time, sometimes even from multiple executables. In this way, graphics hardware can consume multiple vertices, and output multiple fragments, each and every clock tick.

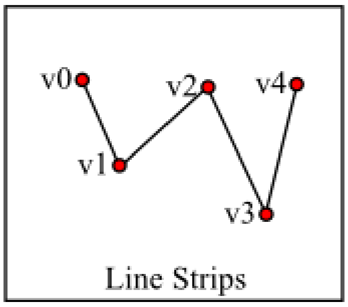
Primitives

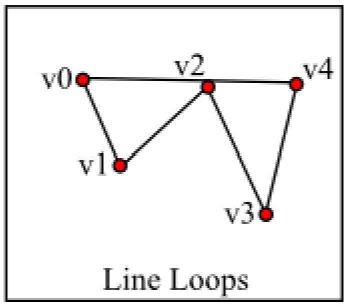
All of the objects you will be rendering on screen, whether they be a simple square, or an entire dungeon's worth of corridors and hallways, are composed of a number of rendering *primitives*. Each of these primitives is made up of a list of vertices, connected together in a variety of ways - except the simplest primitive type, which is simply a 'cloud' of unconnected points. These primitives are used to form a number of object *faces*, sometimes known as *surfaces*, or *facets*.

The exact number of primitives supported is determined by the rendering API chosen - OpenGL 2.0 has the 9 primitives outlined below (along with Quad Strips, which are *very* rarely used), while OpenGL 3.0 onwards actually *removes* the ability to render Quads and Quad Strips, as quads are a trivial case for further decomposition into triangles.

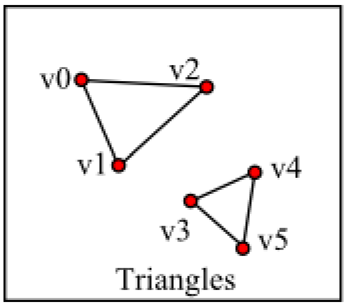
First, let's have a look at the *line* primitives. The simplest form draws a *line* between every pair of vertices, while *strips* draws a line between the current vertex and the previous vertex in a list. *Loops* work like strips, but additionally draws an extra line between the last vertex and the first.

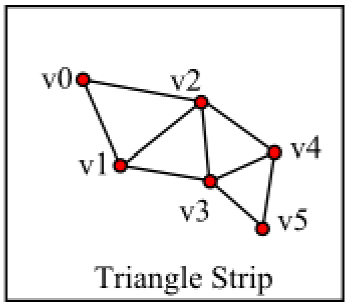


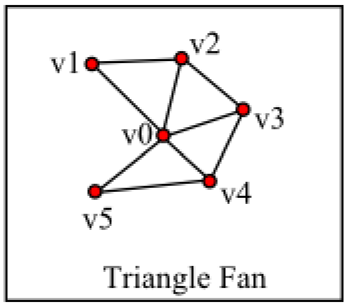




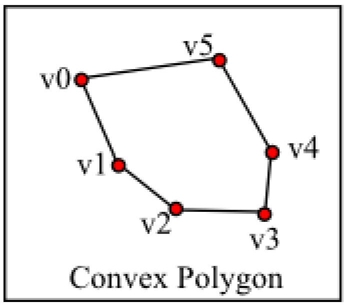
The most common form of primitive is the triangle - graphics hardware is designed around rendering triangles as quickly as possible. The *triangles* primitive simply draws a triangle for every 3 vertices, while *strips* draws an extra triangle for every additional vertex, using the previous two defined vertices as the other triangle corners. *Fans* are a variation on strips, where the first vertex in a list of vertices is always used as a corner, with the triangle completed by the current and previous vertices.

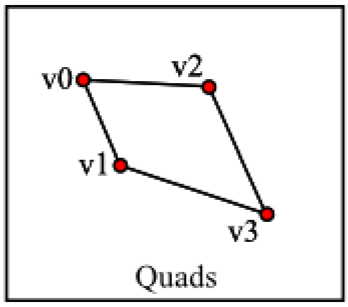


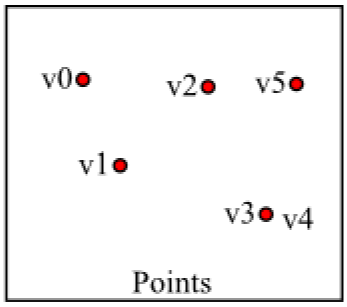




Lastly, we have *points*, *quads*, and *convex polygons*. This last type is limited to convex polygons (i.e. those that that have no interior angles greater than 180°) as they can be easily turned into a number of triangles with the workings of the API, simply by adding an extra vertex to the middle, connected to every point and it's neighbour, like the fan primitive.

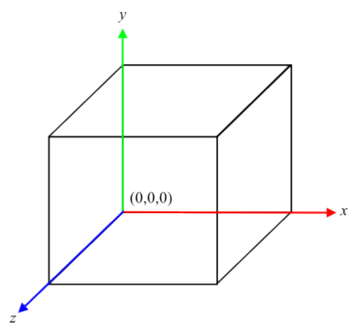






Space

In the process of rendering geometry to screen, it is *transformed* into a number of different *spaces* and eventually 'projected' (or flattened) from 3D coordinates to 2D screen positions, usually via a transformation *matrix*. Unless otherwise noted, these spaces are defined in Cartesian coordinates in $\mathbb{R}^3$, with axis' that run from $-\infty$ to ∞:



Local Space: Whenever geometry is defined, whether it be in an array or vertex data, or loaded in from a mesh file, it is defined in local space. Think of this as being a local origin around which a mesh is defined - for example, your 'local origin' is where you are standing.

World Space: In order to create scenes with lots of geometry and movement, we need to know where objects are in relation to each other. This is done so using a 'global' origin, around which objects are placed. To take the previous example one step further, your world space position could be your latitude, longitude and elevation. Vertex positions are *transformed* from being in local space to being in world space, via a *world transform.*

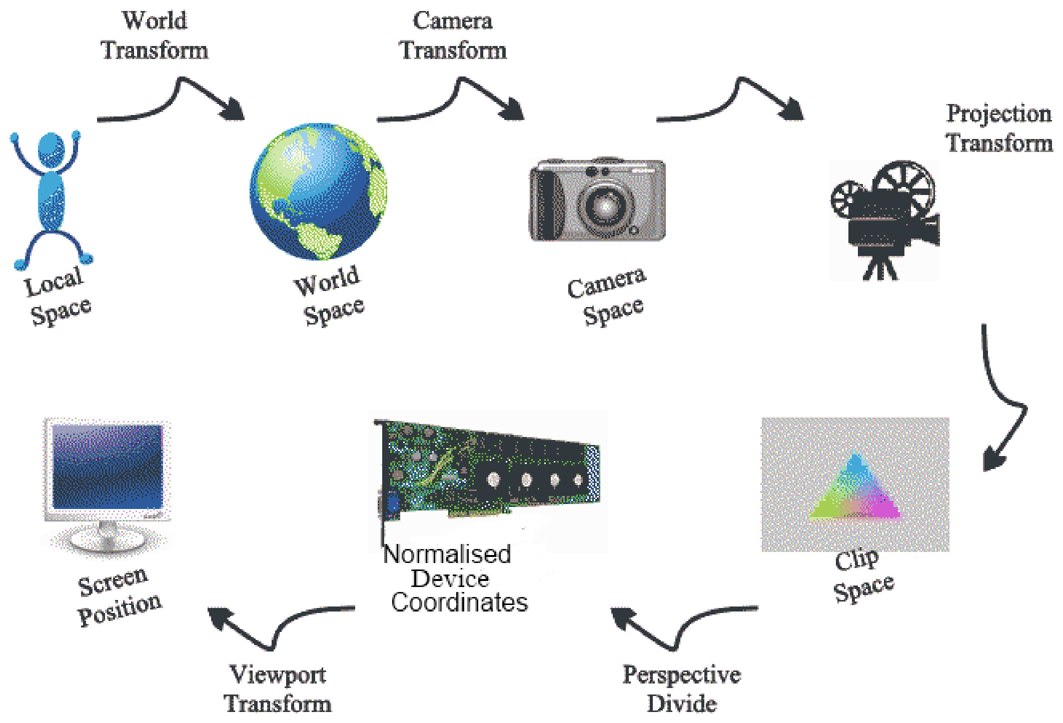
Camera Space: So that vertices can actually be drawn on screen, we need to know where they are in relation to the virtual 'camera' through which we are viewing the scene. This camera has a position in world space, too, so you can think of camera space as having the camera at the origin. Vertex positions can be transformed from world space to camera space via a *camera transform.* One last example involving yourself: If, for some reason, you were in a movie, and were told to stand 10 metres in front of the camera, you'd be transforming yourself to be relative to the camera's position.

Clip Space: Once a vertex position is in camera space, we can apply perspective distortion. We don't want our camera to have 'tunnel vision' - we want to have a field of vision, extending at an angle out from the lens. Our rendering camera can also do something 'real cameras' can't - it can ignore geometry that is too close or too far away via *near* and *far* planes. We can do these things via performing another transform on a vertex position once it is in camera space, the projection transform. You can think of this stage as fiddling with the lens properties of the graphics pipeline's virtual camera. This places vertices into clip space - so called as after this stage, it can be determined which polygons are on screen, and whether they need to be clipped to remove parts that aren't on screen.

Perspective Divide: Objects in the distance look smaller than those close up, but so far, our transforms haven't taken this into account. This is performed by something known as the perspective divide - you'll see why in tutorial 2. For now, just know that this applies the forshortening effect that makes objects in the distance smaller.

Normalised Device Coordinates: Once these steps have been performed, the vertex data is in Normalised Device Coordinates - ones which can be operated upon by the graphics hardware to perform the actual rasterisation of your vertex data into an image on screen.

Screen Position: Finally, your geometry appears on screen, according to the viewport transform - which includes the position of the rendering window on screen, and its size. Unlike the other spaces, this is a 2D space, with the origin at the corner of the screen - which corner is API dependent, as OpenGL uses the bottom left, while DirectX uses the top left.



APIs

When it comes to rasterised graphics, there are two 'big' APIs - Microsoft's DirectX, and OpenGL, which was originally developed by Silicon Graphics Inc, and is now managed by the Khronos Group. They both essentially do the same thing, rasterise vertex data on screen using matrices and shaders.

DirectX

DirectX was first released in late 1995, and has since reached several milestone releases - the most popular versions of DirectX today are DirectX 9, released in 2002, and DirectX 10, released in 2006. Strictly speaking, DirectX is a series of APIs covering sound (DirectSound), networking (DirectPlay), input (DirectInput) and graphics (Direct2D and Direct3D), but the term DirectX has long since become synonymous with graphics rendering. Generally, a new release of DirectX coincides with a new feature set in graphics hardware - DirectX 8 was thefirst to bring shader capabilities, 9 brought high-precision textures and a more complete shader language, while 10 brought with it geometry shaders. Like DirectX, it has constantly evolved as graphics hardware has improved.

OpenGL

The history of OpenGL goes back to the early 90s, when Silicon Graphics Inc released a subset of their IRIS GL product as an open standard. Due to this openness, OpenGL has become widely adopted across many platforms, including all-software renderers like Mesa3D (http://www.mesa3d.org/). Unlike the strictly defined featuresets of DirectX, OpenGL supports vendor-specific API calls called extensions, allowing new hardware features to be exposed without waiting for a new version of OpenGL to be released.

OpenGL 1 defined the base level OpenGL API structure, allowing vertices to easily be sent to graphics hardware for processing, while OpenGL 2 added support for shaders to the OpenGL standard, via the C-like GLSL language, and the ARB assembly language. GLSL was eventually updated to be capable of running geometry shaders as OpenGL 3 took shape. Another important feature of OpenGL 3 was the removal of lots of old OGL 1 and 2 features, and moving to a fully programmable graphics pipeline. This was achieved via the definition of two profiles - the core profile, which stripped away old features that could be performed within shaders or were unlikely to be hardware accelerated, and the compatability profile, which kept parts of the old API active for legacy applications.

The introduction of the OpenGL 4 API brought OpenGL up to feature parity with DirectX 11, enabling the use of hardware tessellation, via tessellation control shaders. It also includes features like per-pixel counters and byte packing of data to improve bandwidth.

Choosing an API

OpenGL and DirectX are pretty similar in terms of feature sets these days, however only OpenGL is properly cross platform. Windows, Linux, and the PlayStation 3 all use (or can use) OpenGL of some sort or another, while DirectX is limited to Windows and Xbox. Most mobile devices, and lately web browsers, also support hardware accelerated graphics rendering via a subset of OpenGL, known as OpenGL ES, making OpenGL an obvious starting point for the aspiring graphics programmer. In this tutorial series graphics programming will be introduced via OpenGL - more specifically OpenGL 3, via its core profile. This allows a high degree of graphics programming via shaders, removing the need for lots of 'legacy' OpenGL API calls - and ensuring that you learn the theory of graphics programming, not strictly OpenGL programming. But why aren't we using OpenGL 4 for this tutorial series? A quick look at the Steam hardware Survey reveals that only 5.56has an OpenGL 4 compatible graphics card - the lab machines may have high-end graphics cards, but your end users likely don't. However, if you want to experiment with OpenGL 4 in your own projects, then feel free!

The OpenGL Rendering Pipeline

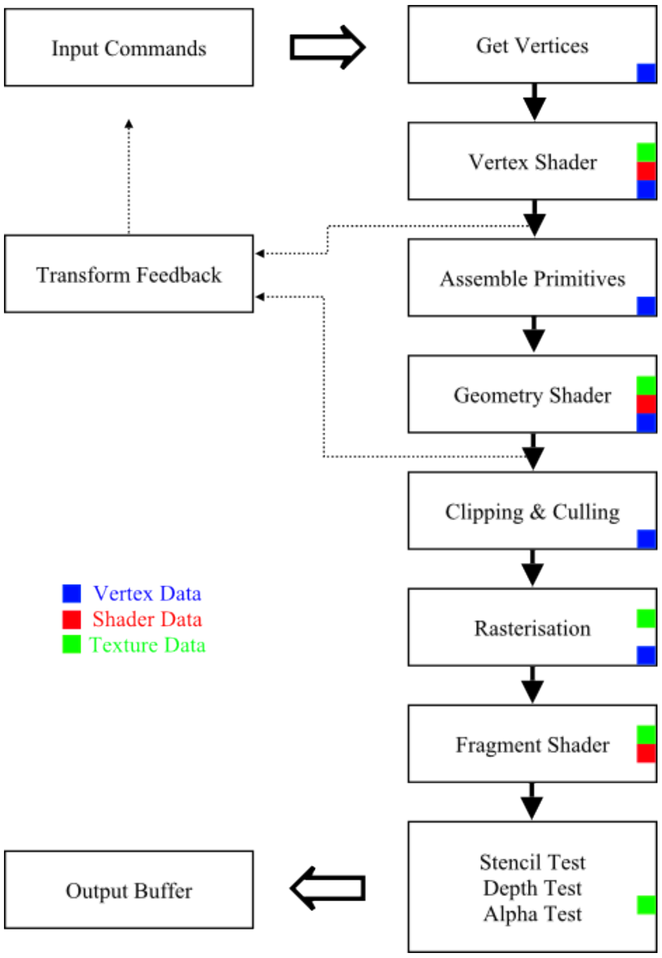
OpenGL works by sending streams of graphics rendering commands to your graphics hardware, one at a time, in the order you call the API functions in. Here's a classic example of OpenGL 1.x programming:

```
 1 glColor3f (1.0 ,0.0 ,0.0) // Set the draw colour red
 2
 3 glBegin ( GL_TRIANGLES ) // Let 's draw a triangle primitive !
 4    glVertex2f (0.0 , 0.0) // left corner
 5    glVertex2f (1.0 , 0.0) // right corner
 6    glVertex2f (0.5 , 1.0) // top corner
 7 glEnd () // Had enough of triangles now ...
 8
 9 glBindTexture ( GL_TEXTURE_2D , 1) // Draw using a texture map
10
11 glBegin ( GL_TRIANGLES ) // Let 's draw a triangle primitive ...
12    glVertex2f (10.0 , 0.0)
13    glVertex2f (11.0 , 0.0)
14    glVertex2f (10.5 , 1.0)
15 glEnd ()
```

OpenGL Commands

This will tell OpenGL to draw things red (line 1), then to begin drawing triangles, passing 3 vertices to be rendered on screen. Then, a texture is bound, and another triangle is drawn. It's important to note that in OpenGL, API calls are not 'blocking', so the glEnd API call won't 'wait' for the triangle to be drawn before returning. The commands are simply 'pushed' onto a FIFO queue, which are 'popped' off one after another by the graphics card driver, and sent to the hardware - drawing and state change calls are guaranteed to be sent to the graphics hardware in the correct order. OpenGL works as a massive 'state machine' - state changes such as colour setting stay 'set' until another command changes the same state. Take the above code example - the second triangle will actually be drawn red, due to the previous state changing call glColor3f. If a third triangle were drawn, it too would have a red colour, and a texture applied to it, unless the colouring or texturing states were changed.

But how does OpenGL and your graphics hardware interpret these input commands? How do they turn into a final image? Well, input commands directly related to drawing vertices on screen are pushed through a graphics pipeline, a set of stages which work on graphics data according to the input commands they receive. In older versions of OpenGL, this graphics pipeline had a stage for every graphics operation - there was a stage to discard fragments below a certain alpha value, to add a fog value, and so on. Later versions, including OpenGL 3.2 used in this tutorial series, do away with this 'fixed function' pipeline, with most functionality set within programmable shaders instead. The OpenGL 3+ graphics pipeline has the following stages:



The graphics pipeline of OpenGL 3.2

The vertices required to render the current object are processed by the currently set vertex shader, before being assembled into groups of primitives. These primitives can then be enhanced or modified by the geometry shader, before either being culled if they are off screen, or clipped, if they are partially on screen. Then, the actual process of rasterisation takes place, determining which pixels a primitive takes up on screen, and running a fragment shader to determine its final colour on screen.

With the fragment shader completed, post-shader processing such as stencil testing (limiting drawing to certain pixels), depth testing (only drawing to the current fragment if the current primitive is in front of what is already on screen) and alpha blending (mixing colours of transparent objects) is performed, and the final value (if any) written to the screen buffer.

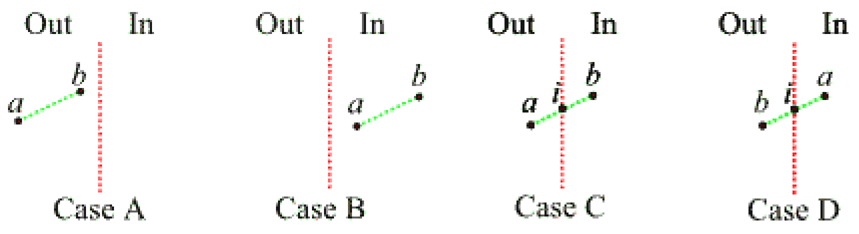
Of note is an optional stage, the *feedback buffer*. This allows the post-transformation state of vertices to be saved out to a data buffer. This is useful, both for debugging purposes, and for saving processing time by not using complex vertex shaders for multiple frames in static scenes.

Clipping

Earlier, it was mentioned that triangles may need to be *clipped* to remove parts that aren't visible. This is to reduce the cost of rasterising the triangle into the final image on screen. Imagine a triangle with edges a hundred miles across each edge, but with only the very corner visible on screen - that's a whole lot of triangle surface area that will be processed, despite never being seen on screen. To alleviate this, triangles that have vertices positioned off the edge of the screen (but which are still visible on screen) are cut up into smaller triangles that fit exactly on screen. There are a number of different algorithms for clipping a triangle, many of which can handle arbitrary clip areas and polygons, but the most common is known as *Sutherland-Hodgeman* clipping. This method takes each edge of the screen in turn, and 'cuts' the edges of the polygon to be clipped, adding vertices to a list according to 4 simple case rules:

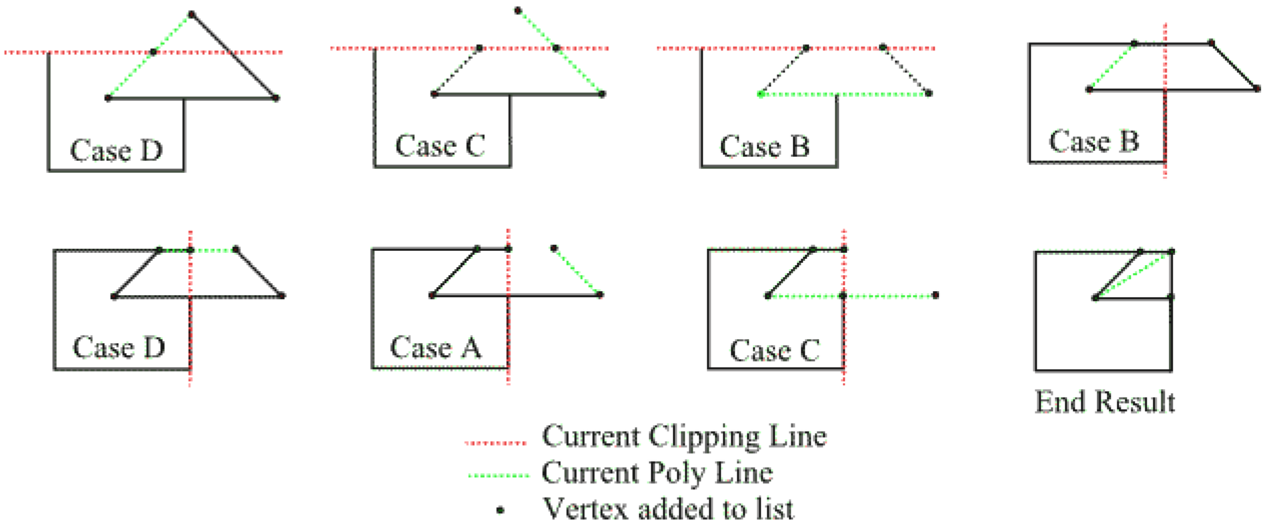
Case A: If both vertices of the current polygon edge are *outside* the current edge, neither vertex is added to the output list.
Case B: If both vertices are *inside* the current clipping edge, vertex b is added to the output list.
Case C: If vertex a is *outside*, and vertex b *inside* the clipping edge, the intersection vertex i is calculted, and the vertices i and b are added to the output list.
Case D: if vertex a is *inside*, and vertex b is *outside* the clipping edge, the intersection vertex i is calculated, and added to the output list.

The vertices left in the output list are then clipped against the next screen edge, and so on until only clipped edges that are visible on screen are left.



The four cases of Sutherland Hodgeman polygon clipping

Here's an example to demonstrate how Sutherland-Hodgeman can clip a triangle to the screen edges:



The result of clipping a triangle against the top and right edges of the screen.

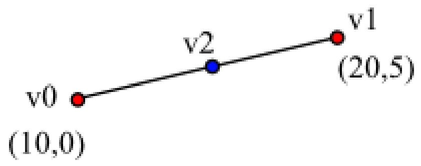
First off, the top edge of the screen clips the 3 edges of the triangle, resulting in a trapezoid. This shape is then clipped by the right hand screen edge to a simpler quadrilatteral shape, which is fully inside the bottom and left screen edges. This shape can then be easily triangulated into two triangles.

Interpolation

The fragment shader stage of the pipeline uses vertex data as its input - but to generate a correct image, this data is interpolated between the vertices that result in a particular fragment. All of the vertex data attributes for a particular vertex are interpolated - positions, colours, and any other vertex data are all interpolated to generate the final image. For example, imagine a simple line made of two vertices - one red and one green. As this line is rasterised, the line would create fragments like so:

As rasterised fragments get closer to the end of the line, they are progressively more influenced by the green vertex, and less by the red vertex.

Interpolation of a vertex attribute value along a line is pretty easy to calculate. For example, imagine we have two vertices *v0* and *v1*, and we want to find the position of the point *v2*, which is exactly half way between *v0* and *v1*.

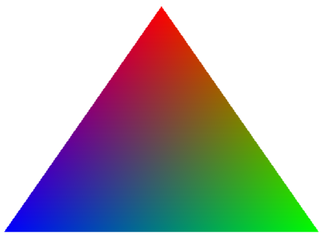


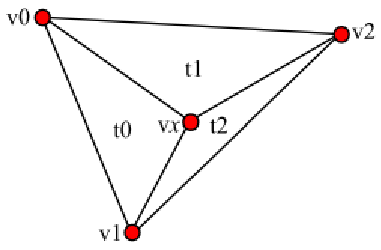
We can find the x and y values of v2 like so:

$$v2_x = (v0_x \cdot (1 - \mu)) + (v1_x \cdot \mu)$$
$$v2_y = (v0_y \cdot (1 - \mu)) + (v1_y \cdot \mu)$$

Where μ (the Greek letter *Mu*) is a value between 0 and 1 - in this case we want to find the position exactly half way between v0 and v1, so a μ of 0.5 is used, giving the position of v2 as (15, 2.5). This is an example of *barycentric* coordinates, which defines a series of weighted points (in this case vertex positions), weighted by coefficients which add up to 1 (in this case μ and (1 - μ).

Interpolation can also be performed on other primitive data. For example, here's a triangle, with red, green and blue vertices:





Left: A triangle with interpolated colours Right: The sub-triangles formed from interpolated point vx

From these vertices, we can work out the exact interpolated colour of any point within the triangle, again using barycentric coordinates. This time, instead of using μ, we can find an interpolated value for the point *vx* using the barycentric weights α, β, and γ (*Alpha*, *Beta*, and *Gamma*).

$$vx = (v0 \cdot \alpha) + (v1 \cdot \beta) + (v2 \cdot \gamma)$$

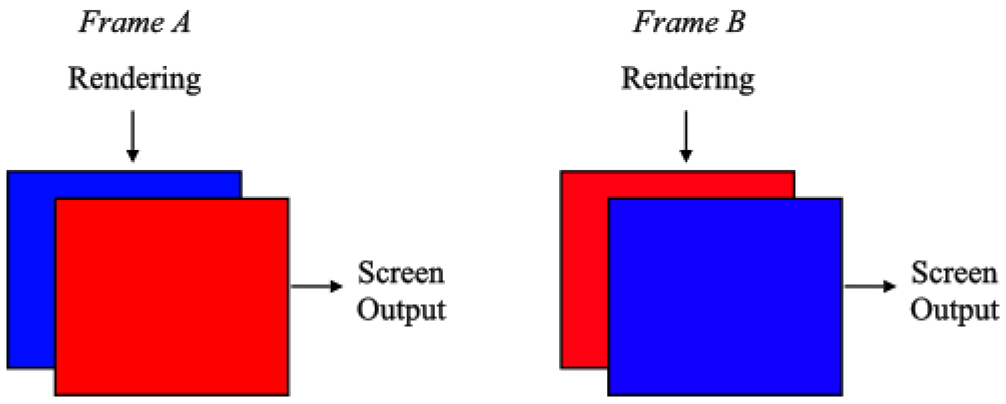
But what are the values of α, β, and γ? For any triangle *t*, and a point *vx* (the position we want to find the interpolated value for), we can split triangle into 3 'subtriangles' t0, t1, and t2. From these subtriangles, α, β, and γ can be worked out using the ratio of their area compared to the area of the original triangle *t*.

$$\alpha = Area(t2)/Area(t)$$
$$\beta = Area(t1)/Area(t)$$
$$\gamma = Area(t0)/Area(t)$$

Buffering

When rendering, you don't directly access the screen; rather, you are rendering into a *buffer* - a block of memory large enough to store enough information for each pixel. Depending on the rendering technique, you may have a single buffer, or multiple buffers - usually two. In single buffering, the contents of the buffer is constantly sent to the screen (known as *blitting*). This poses a problem - the data you are sending to screen is being updated constantly, as you render new objects. The side effect of this is that you can see the screen flickering, as the colour data of the buffer changes due to new objects being rendered, or the screen cleared.

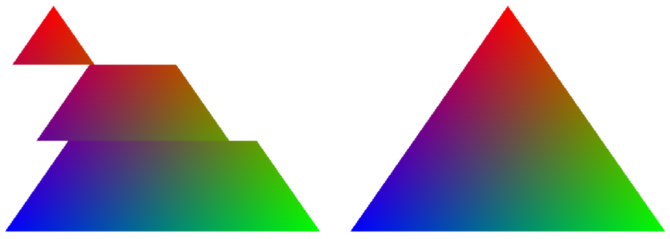
The solution to this is to have *multiple* buffers. When using multiple buffers, one buffer is used to draw into, while another is blitted to the screen. Then, when the drawing for a frame is completed, these buffers are *flipped*, presenting the whole rendered scene to the blitter at once, and providing another buffer to draw into. Doing so delays the update of the screen by a frame - you will always see the *previous* frame's rendering. In such multiple buffering techniques, the buffer being rendered on screen is known as the *front* buffer, while the buffer being drawn into is the *back* buffer.



An example of double buffering - multiple buffering using two buffers. In frame A, the blue buffer is used to draw into, while the red buffer is blitted to the screen. In the next frame, the buffers are flipped, so that the blue buffer is presented on screen, and the red buffer used to draw into. In the following frame, the buffers would be flipped once more, and so on.

Vertical Sync

When a frame's rendering is completed, the buffers are swapped, and the screen updated from the new front buffer. However, what if the buffer flips are quicker than the refresh rate of the monitor? Computer monitors can only refresh their image so many times a second, usually somewhere between 60 and 120 times a second. If a monitor refreshes its image 60 times a second, and the flip rate of the rendering updates is out of sync with this (i.e. it is rendering faster or slower than 60FPS), *screen tearing* occurs. If objects are moving on screen in such a non-synchronised update, they will appear to 'tear', appearing in multiple positions on screen. The solution to this is to use 'vertical sync', where the flip between buffers is deferred until all of the previous buffer is completed. This limits the framerate to a multiple of the screen refresh rate, but prevents tearing, so many APIs have this *v-sync* capability enabled by default.



Left: A moving triangle with no v-sync Right: How it should look, correctly synchronised.