3D Graphics Library

Technical Design Documentation

100161995

12/16/2011

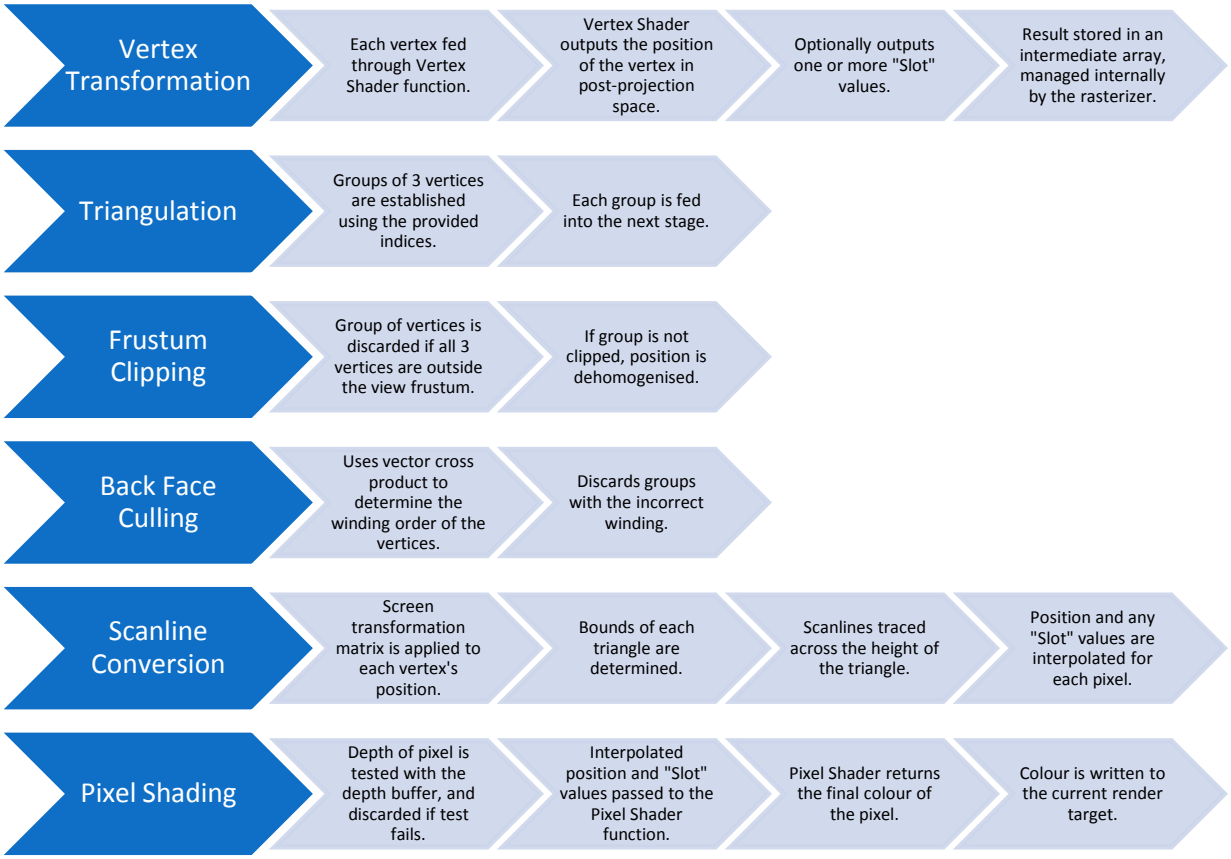
Table of Contents

Rasterizer Design

At the core of the graphics library is the Rasterizer class. It is responsible for the rendering of triangles to the current render target, using functions provided by the calling application to determine the position of each vertex on the target, and the ultimate colour of each pixel. The Rasterizer has been carefully designed to provide the most flexibility whilst maintaining performance and code reuse.

Vertex and Pixel Shader Pipeline

The core design of the Rasterizer has been modelled on existing hardware graphics pipelines. The design incorporates the idea of vertex and pixel shader functions. These allow the calling code to completely customise the rendering process, making the implementation of various lighting models and rendering techniques very simple. With this pipeline, it is possible to implement Flat, Gouraud or Phong shading by simply providing a different vertex and pixel shader for each technique. The following diagram outlines the main stages of rendering within the Rasterizer.



Member Function Pointers

The vertex and pixel shader functions are implemented using C++ member function pointers. Two types are defined in `Rasterizer.h` which provide the interface between the functions and the Rasterizer:

```
typedef void (DemoEngine::*VertexShaderCallback)(const Vertex& input,
VS_Output& output) const;
typedef void (DemoEngine::*PixelShaderCallback)(const VS_Output& input,
Vector4& output) const;
```

These functions are declared as `const` as they should not cause any lasting effect when called.

Vertex Shaders and Slot Values

At the first stage of the pipeline, the full list of vertices for a specific model is fed into the Rasterizer. The Rasterizer then takes each vertex, and calls the Vertex Shader function provided, passing the current vertex by reference. The Vertex Shader must transform the vertex's position into post-projection space, but must not perform the division by W. This is achieved by transforming the position by the combined world view projection matrix. The result of this transformation must be stored in the Position member of the VS_Output structure.

The VS_Output structure also contains a static array of Vector4 types, named "Slots". These slot values have been modelled on the use of "TexCoords" within DirectX HLSL shaders. Each slot acts as a general purpose interpolation slot. Values placed in these slots will be interpolated between each neighbour vertex during scanline conversion. The result of this interpolation is passed to the pixel shader in the same slot index.

As these slots have no specific requirement, any value can be interpolated across each triangle and used inside the pixel shader for any purpose. For example, during Gouraud shading, lighting calculations are performed in the vertex shader, and the resulting colour is returned in the first slot. This colour is automatically interpolated by the Rasterizer, and then drawn to the render target by a pixel shader that simply returns the value from the first slot. Conversely, during Phong shading, the slots are used to interpolate world position and normal values, allowing the light calculations to be performed per pixel, inside the Pixel Shader function. This design allows the Rasterizer to be extremely flexible, as lighting and shading models are implemented separately from the Rasterizer.

The result from each call to the Vertex Shader function is stored in an intermediate array managed by the Rasterizer. This array provides a mapping between the original model vertices to the transformed VS_Output structures, which allows the use of the model's indices to create triangles from the transformed vertex data.

Frustum Clipping

The triangulation stage is responsible for grouping VS_Output structures into triangles. These triangles then enter the frustum clipping stage. At this stage the position value of each vertex is homogenous. It is now possible to determine if a vertex is within the camera's view frustum by comparing the X, Y, and Z values of position against the W value.

The projection matrix used during vertex transformation has the effect of mapping the camera's view frustum to a cuboid, with the centre of the near plane placed on the origin. The left and right planes lie at X = -1 and X = 1 respectively, and the bottom and top planes lie at Y = -1 and Y = 1. The near plane lies at Z = 0, and the far plane lies at Z = 1. It follows that for a point to be within the view frustum of the camera prior to the projection transformation, it must satisfy the following inequalities after the transformation:

$$\begin{pmatrix} -1 \leq x/w \leq 1 \\ -1 \leq y/w \leq 1 \\ 0 \leq z/w \leq 1 \end{pmatrix}$$

By multiplying by w, these inequalities become:

$$\begin{pmatrix} -w \leq x \leq w \\ -w \leq y \leq w \\ 0 \leq z \leq w \end{pmatrix}$$

These inequalities allow us to determine if a point is within the view frustum prior to the division by W. Not only does this allow us to avoid an unnecessary division on vertices lying completely outside the frustum, it also avoids the need to perform point-to-plane intersection tests in world or view space, allowing a more streamlined design for the Rasterizer, as knowledge of the camera settings are not required to perform view frustum clipping.

This test is implemented within the `bool Vector4::IsInView(void) const` member function, which is called by the Rasterizer during the view frustum clipping stage. Triangles which do not have at least 1 vertex within the view frustum are discarded. If the view frustum test is passed by a triangle, each of its vertices' position values are dehomogenised, then passed to the next stage of the pipeline.

Back Face Culling

At this stage of the pipeline, triangles which do not face the camera are discarded. This is achieved through the use of a vector cross product. Firstly, two vectors are determined by subtracting the first vertex's position from the positions of the other two. The cross product of these two vectors is then calculated. As the vertices are now in post-projection space, the camera is facing directly down the Z axis from the origin. The vector cross product results in a vector with either a positive or negative Z value. This determines if the triangle is facing towards or away from the camera, based on the winding order of the vertices.

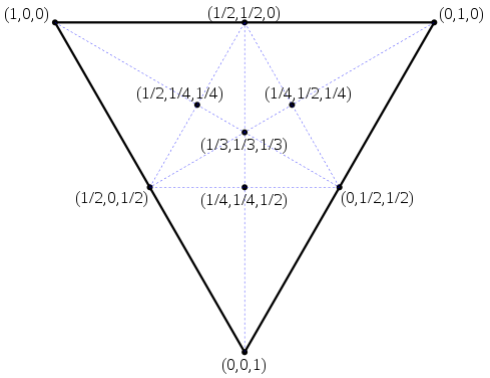
The Rasterizer supports the culling of both clockwise and counter clockwise faces. Triangles are discarded if the Z value of their cross product is either greater than or less than 0, depending on the current render settings.

Scanline Conversion

At this stage of the pipeline, only triangles which are visible within the view frustum are left. A screen transformation matrix is applied to each vertex to map the triangle onto the screen. This matrix is constructed from the dimensions of the current render target, and maps the projection cuboid to the bounds of the render target.

The rendering pipeline now splits into two separate, but similar stages based on the current render fill mode setting. In "Solid" mode, the Rasterizer is instructed to fill the triangle primitives with pixels, using scanline conversion. In "Wireframe" mode, the Rasterizer will trace the outlines of each triangle using the Bresenham Line Algorithm. Regardless of fill mode, the Rasterizer performs interpolation of the position and slot values for each vertex, and then uses the current Pixel Shader function to determine the final pixel's colour.

In Solid fill mode, interpolation is performed using Barycentric Coordinates. These coordinates represent points within a triangle as 3 values. Each value determines how "close" the point is to the associated vertex. The diagram to the right shows the Barycentric coordinates for various points within an equilateral triangle.



(7)

During the scanline conversion, values for the first two coordinates, named u and v, are determined for each pixel. These two values are used to interpolate the position and slot values using the following equation where values A, B and C are values from each of the three vertices.

$$result = Value_C + (Value_A - Value_C)u + (Value_B - Value_C)v$$

Care has been taken to ensure that rounding errors do not occur during the scanline conversion. These errors can cause issues at boundaries between triangles where either a pixel is rendered more than once, or a pixel is not rendered at all, leaving a black line between the boundaries of a triangle mesh. The scanline algorithm has been designed to adhere to the rules outlined in (1), such that these errors are minimised.

In Wireframe fill mode, a simpler interpolation method is used. A parametric value "t" is determined, which describes the distance travelled from the start vertex to the end vertex. This variable takes values between 0 and 1. Interpolation is then performed using the equation of a linear Bezier curve (2):

$$result = (1 - t)P_0 + tP_1$$

The interpolated values generated in Wireframe mode can be passed to the pixel shader in the same way as in Solid mode. This allows the rendering of a "lit" wireframe model, rather than a wireframe of a solid colour, since the same Pixel Shader function will be used in both Solid and Wireframe mode.

Pixel Shading and Depth Testing

The results from the scanline conversion stage are passed to the final stage of the pipeline. The pixel shading stage first performs a lookup in the current depth buffer. The depth buffer is an array of floating point values, one for each pixel. It stores the depth value of the closest pixel to the camera at each pixel location. This value is obtained from the interpolated vertex positions' z values.

If the depth of the current pixel is less than or equal to the depth stored in the depth buffer, then this pixel is not currently occluded by other geometry. This pixel passes the depth test and is allowed to be rendered to the current render target.

If however, the depth of the current pixel is greater than the depth stored in the depth buffer, this pixel is occluded by some other geometry that has already been rendered to the render target. This pixel will fail the depth test and is discarded by the Rasterizer.

This method of per pixel depth testing eliminates the requirement for depth sorting the triangles before the rendering stage. It also means that depth information persists between multiple rendering tasks, allowing multiple models to be rendered without depth occlusion issues.

If the depth test is passed, the new depth value obtained from the interpolated vertex positions is written to the depth buffer, and the pixel is shaded using the Pixel Shader function. The Rasterizer calls the Pixel Shader function passing the interpolated VS_Output structure by reference as input. The result of the Pixel Shader is returned as a Vector4 structure, where each component represents a single colour channel (Alpha, Red, Green and Blue). This value is then clamped between 0.0 and 1.0 for each component, multiplied by 255, then cast to byte values. These byte values are shifted and logical OR'ed together to form the final ARGB colour DWORD value for the current pixel. This value is then written to the current render target.

Perspective Correction

The interpolation of values across triangles can introduce artefacts caused by the perspective projection. To avoid this, perspective correction needs to be applied to the interpolated values. As outlined in the module tutorial documents, this is achieved by first calculating the reciprocal of the z depth value, and the result of dividing the values to interpolate by the z depth value. These values are then interpolated instead, and the final value is obtained by performing a second division by the reciprocal of z. This technique is known as Hyperbolic Interpolation.

This technique is used to correct issues in texturing caused by the perspective transformation. However, it is not enough to only perform hyperbolic interpolation on texture coordinates. From my own findings, all values interpolated across triangles are subject to distortion from perspective, and thus must all have perspective correction applied. This is achieved in modern graphics hardware through the use of the "TexCoord" HLSL semantics, which are interpolated across each triangle using hyperbolic interpolation. As the "Slots" values within the VS_Output structures work in a similar way to "TexCoords" on graphics hardware, applying perspective correction is a relatively simple process, and completely transparent to the vertex or pixel shaders.

As a result, the Rasterizer does not directly interpolate "slot" values across each triangle. Instead, each vertex's values, returned from the vertex shader function, are first divided by the vertex's position's W value (which contains the original depth of the vertex). It is these values that are interpolated across the triangle. The reciprocal of z is stored in the W component of each vertex's position, just after the screen transformation, as this value is now useless (it always contains 1.0). At each pixel, the interpolated values are then divided by the interpolated reciprocal of z to restore the final value, which is passed to the Pixel Shader function. This allows all values in the Slots array to be corrected for perspective, without any further effort from the vertex or pixel shaders.

Render Targets and Depth Buffers

Advanced graphics techniques, such as shadow mapping, require the use of off-screen buffers to store intermediate data used when rendering the scene. To achieve this, the concept of separable render targets and depth buffers has been used.

A Render Target is an area of memory representing a colour bitmap. The RenderTarget class creates and manages a GDI bitmap section with the specified width and height. Render targets are used as drawing buffers and can be copied to the screen when drawing is complete.

A Depth Buffer is an area of memory used to store the depth of each pixel from the camera. The DepthBuffer class manages an internal 2D array of floating point values with the specified width and height. Depth buffers are used during the rendering process to determine if a particular pixel is occluded by another, preventing geometry from drawing over objects that are closer to the camera than it is.

The Rasterizer supports switching of the current render target and depth buffer between draw calls. This allows advanced techniques to be achieved. For example in shadow mapping, the scene is first drawn from the light's point of view. The render target is set to NULL, as no pixel data is required. This skips the pixel shader stage of the pipeline. The depth buffer is set to a specific buffer used to store the shadow map. After the rendering process, the shadow map depth buffer contains the depths of all pixels closest to the light, from the light's point of view. The application can now switch the render target and depth buffer back to the original ones, to render the scene from the camera's point of view. During this process, the depth of each pixel from the light is calculated and compared with the stored depth in the shadow map depth buffer. If the calculated depth value is greater than the depth value in the buffer, the current pixel is in shadow; otherwise the current pixel is in light. Using this information, dynamic shadows can be rendered.

When the render target and depth buffers are changed, the Rasterizer recalculates the screen matrix required to map vertices to the correct locations on the targets. This allows the Rasterizer to support any dimension of render target and depth buffer.

Removal of GDI+ Fill Polygon

As a result of the vertex/pixel shader pipeline design, and the use of per-pixel depth buffering, it became impractical to support the GDI+ Fill Polygon routine for Gouraud shading. The depth testing would need to be replaced with depth sorting, as a special stage in the pipeline. Furthermore, support for multiple models would be lost, as the depth sorting procedure only takes into account the geometry of a single model at once. The GDI+ Fill Polygon routine would also be incompatible with the multithreaded nature of the Rasterizer, and would break the design of the interpolation slots used in the VS_Output structures.

Due to these reasons, support for the GDI+ Fill Polygon routine was removed from the Rasterizer. All solid fill rendering modes utilize the custom fill polygon function defined in the Rasterizer class. The GDI+ library is now only used to import bitmap files as textures, and to draw the HUD text to the screen.

MD2 Model File Loader

Texture Coordinate Issue

Code for an MD2 Model File loader was provided as part of the module specification. However, this was very simplistic and the vertex structures it generated were not compatible with the vertex/pixel shader architecture of the Rasterizer. The issue lies in the use of texture coordinates. Within the MD2 file format specification, texture coordinates are stored on a “per face” basis. However, to utilise MD2 models within the Rasterizer, texture coordinates (and all other vertex properties) need to be declared on a “per vertex” basis.

To achieve this, the MD2 file loader code was modified to perform a post processing stage to each face and vertex as each frame is loaded from the MD2 file.

Each frame’s vertices and faces are first loaded into an array of intermediate structures, named MD2Temp_Vertex and MD2Temp_Face respectively. Each face structure contains a one-to-one map of index numbers to vertices which form that face. Each vertex contains a one-to-many map of texture coordinates to faces. Once both arrays have been initialised, it is possible to determine which vertices have multiple texture coordinates defined for it. Since a vertex can only store one texture coordinate, vertices with multiple coordinates must be split apart, with one vertex per texture coordinate.

The loader then begins to insert additional vertices for each vertex that contains multiple texture coordinates. Each copy is assigned a single texture coordinate from the list of coordinates the original vertex owned. The loader uses the one-to-one index to vertex mapping, in each face, to reassign index numbers for each face affected by this insertion. The final texture coordinate for each vertex is stored in the “UnifiedTexCoord” member field.

Once all the vertices have been reassigned, the loader copies the relevant information from each temporary vertex structure to the final vertex array, calculating face normals using vector cross products in the process.

Originally there was an issue with triangle winding orders. The C++ Standard Template Library’s implementation of map<> does not preserve the insertion order of elements. Instead, keys are retrieved in ascending order, which caused issues when remapping faces as they depend on the original order of the texture coordinates. To resolve this, a very simple OrderedMap<> template class was written which maintains the insertion order using a vector<> type to store the key and value pairs, and a map<> type to store lookup indexes to the vector. Methods such as Remove() etc, are not implemented as they were not needed within the MD2 Loader.

Animation

The MD2 Loader was also extended to handle key frame animation. Instead of reading a single frame from the file, all the frames are read from the file and stored in an array within the Model class. At each game update, these frames can be iterated over to give the impression of character animation.

Texturing and Materials

Texture Memory Format

Textures can be loaded from both standard bitmap files, and from MD2 compatible PCX files. In both cases, the texture data is stored in memory as a two dimensional array of Vector4 objects. Each component of the Vector4 objects represents the four colour channels of the texel's colour. The values of these channels range from 0.0 to 1.0. For example opaque medium grey would be represented as (0.5, 0.5, 0.5, 1.0). This format works very well with the lighting calculations within the pixel shaders, as colours can be combined by simply multiplying or adding two Vector4 objects, depending on the lighting model in use.

Texture Filtering

Textures are sampled on a per pixel basis, inside the pixel shader functions, using the interpolated texture coordinate value passed to the shader from the Rasterizer. The simplest form of texturing involves selecting the single pixel that a given texture coordinate relates to. Whilst this simple technique is very fast, it also creates aliasing. Areas of detail on the textures can appear "blocky" and pixellated. This can be prevented through the use of texture filtering.

The graphics library currently implements bilinear texture filtering (3). This technique involves sampling four pixels rather than one, then blending the results together to cause a smooth transition from one pixel to another. Whilst this computation is more expensive, it results in much smoother textures on objects.

Materials

Each object in the demo application is associated with a material. This is a simple data structure which contains information about the lighting and texturing properties of each object to be rendered. Note that the concept of materials is completely separate to the Rasterizer itself, as the values from the materials are only used within the vertex and pixel shader functions.

The materials structure contains values for ambient, diffuse and specular lighting constants, a specular power value, and a diffuse texture reference. The lighting constants describe the amount of each colour channel that the surface reflects from each light type. By altering these values it is possible to cause a "tint" effect on the lighting results for that object. The specular power value is used in the computation of specular highlights, and describes the "smoothness" of the surface. The higher the value, the smaller the specular highlights will be, which gives the impression of a shinier surface. The diffuse texture reference is used to assign a particular texture to an object. This texture is sampled, using either point or bilinear filtering, to obtain the diffuse colour for a given pixel. This colour is combined with other lighting calculations to determine the final pixel colour.

Multithreading

Modern computer processors have a number of processing cores which allow multiple tasks to run in parallel. This can increase the performance of an application if the tasks it runs can be broken apart into smaller tasks that can run at the same time as each other. In doing so, the application will be able to take full advantage of the multiple processing cores the computer platform has to offer, and achieve a higher throughput.

The task of transforming a long array of vertices, and then rendering triangles formed from those vertices is a prime candidate for multithreading. The Rasterizer has been designed to utilise multithreading to increase the rendering performance. This allows the application to render very complex lighting calculations, whilst still maintaining an acceptable interactive frame rate.

Creation of Worker Threads

When the application is launched, a call is made to the operating system to retrieve the number of logical processors in the system. This number is used as the total number of worker threads to create within the AppEngine class, and is passed as an argument to the AppEngine class's constructor. The AppEngine constructor makes several calls to the CreateThread() function provided by Windows. The threads enter the program through a common entry function, called WorkerThreadEntry, which uses a small data structure passed by void pointer to call into a member function within AppEngine. The threads then enter a while loop, waiting for work to be dispatched by the application's main thread.

Thread Work Items and Synchronisation

Once the AppEngine object is fully constructed, there will be a number of worker threads that are waiting for work to be dispatched by the main thread. The threads are held by the operating system using a semaphore. A semaphore is an operating system service provided by Windows which prevents further execution of a number of threads, until its internal counter is incremented above 0. When the counter is increased, a number of threads waiting on the semaphore object will be released and their execution will continue. When a thread is released, the semaphore's counter is decremented, until it reaches 0 again, and the threads will be blocked again. This semaphore object is used to hold the worker threads until a work item is available. The semaphore's internal counter is then incremented to match the number of worker threads. This releases all of the waiting threads, and resets the count to 0. Once the worker threads have completed their task, they will be blocked by the semaphore object again, until the next work item is dispatched.

Once the main thread has dispatched a work item to the worker threads, it must also be suspended until the work is completed. This is achieved through the use of an event object, another synchronisation primitive provided by the operating system. Events exist in either the "set" or "unset" states. Threads are held if the state is "unset", and released if the state is "set". Once the main thread has released the semaphore, it then waits on the event object to be signalled. This suspends the main thread. Once the final worker thread has completed the given task, it signals the

event object by calling the SetEvent() method. This changes the event's state to "set", allowing the main thread to continue execution.

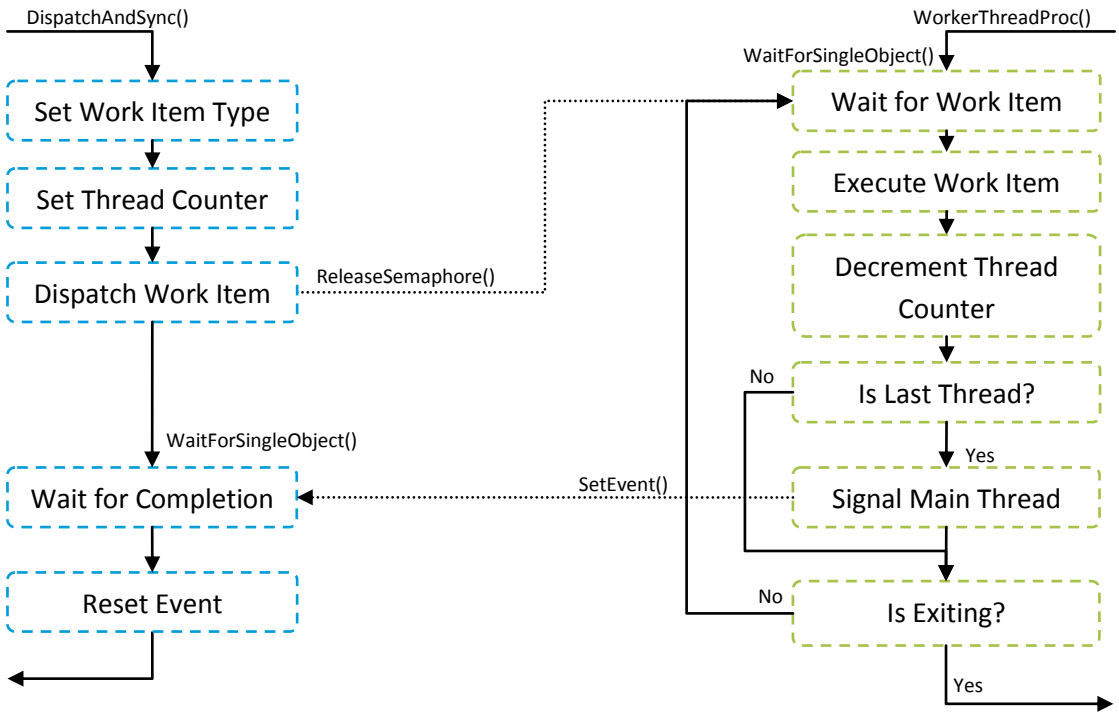
This entire synchronisation process is handled internally by the AppEngine class, via the DispatchAndSync() method. The method uses an enumeration, passed as an argument, to signal to the worker threads which task should be executed. This enumeration value is copied to a shared member variable to be accessible to the worker threads. A counter variable, called "runningWorkerThreadCount" is set to the number of worker threads, then the main thread calls ReleaseSemaphore() to release the waiting worker threads. It then suspends itself by calling WaitForSingleObject() to wait on the event object for task completion.

The worker threads use the enumeration value, stored in the shared member variable, within a switch block to execute the required function. Once the work within those functions is completed, the function returns. Each thread then decrements the "runningWorkerThreadCount" variable. If the result of the decrement is equal to 0, that thread was the last thread to finish working. This thread then signals the event object using SetEvent() to release the application's main thread.

The main thread now resets the event object to the "unset" state and returns from the DispatchAndSync() function. The work item is now fully completed and the AppEngine is in a state to accept a new work item. From the calling code's point of view, the DispatchAndSync() method runs synchronously, i.e. it only returns once all the work is complete.

The worker threads continue to wait for the semaphore object in a loop, until the "Exit" work item is dispatched by the main thread. This is a special work item used to terminate the worker threads. It is dispatched from within the AppEngine's destructor.

This procedure is summarised in the following flowchart:



InterlockedIncrement and InterlockedDecrement

Two types of work item exist within the Rasterizer: Transform Vertices, and Draw Primitives. These work items operate on a batch of vertices or faces stored in an array. In order to delegate a vertex or face to each worker thread, an index into the work item's array is maintained by the Rasterizer. Each thread first increments this index, then retrieves the new, incremented value of the index. It uses this value to look up the particular vertex or face it should be working on, and then executes the required work for that vertex or face.

This allows the Rasterizer to operate on a "first come first served" basis, allocating the next vertex or face in the array to the next thread to ask for it. This design allows for reliable load balancing between each worker thread. If during the Draw Primitives work item, a thread is allocated a particularly large face, this face will take a long time to render. Whilst that thread is rendering the large face, other worker threads can retrieve and render smaller faces in parallel.

However, incrementing or decrementing a member variable in plain C++ can lead to race conditions between threads. This is where two or more threads attempt to increment the value at the same time, leading to unexpected results. It is caused by a particular operation requiring more than one instruction to execute. In order to avoid race conditions, the increment or decrement operation must be "atomic", i.e. cannot be subdivided into more than one step. This ensures that the operation cannot be interrupted by another thread before it has completed.

Both InterlockedIncrement and InterlockedDecrement can alter the value of a variable without causing race conditions. InterlockedIncrement is used within the Rasterizer to obtain the next unique index to the work item's array for each thread. InterlockedDecrement is used within the WorkerThreadProc() to determine when the last thread has finished executing the work item.

Intel Streaming SIMD Extensions

Since the release of the original x86 instruction set, Intel has added many advanced instructions that modern desktop processors support. These instructions are designed to perform certain tasks in a more efficient manner. One set of these instructions are the Streaming Single instruction multiple data Extensions (SSE) (4). These instructions can perform a range of scalar operations such as addition, subtraction and multiplication on up to 4 floating point values at once. This makes them an ideal candidate for optimising the vector and matrix code from the graphics library. The ability to perform the same operation on multiple pieces of data at once allows greater performance than sequentially applying the same operations to each piece of data separately.

__m128 and Conditional Compilation

To access these processor instructions requires the use of special data types and intrinsic functions. The __m128 data type defines the 128 bits of data which makes up four 32-bit floating point values. Instances of this type are passed to various intrinsic functions such as _mm_add_ps().

Initially the vector and matrix math library was written with plain C++. All vector operations were performed on each element individually. Once the plain code had been written and tested, it was possible to extend the vector and matrix classes to utilise SSE instructions.

To preserve the plain C++ vector and matrix code, conditional compilation directives are used in each of the vector and matrix class's functions. By creating two new solution configurations (DebugSSE and ReleaseSSE), and setting the compiler settings to define the symbol "USE_SSE", it is possible to switch between SSE and non-SSE builds. This proved very useful whilst testing the SSE extensions.

Unions and Use of Public Fields

To allow access to the individual elements of the __m128 data type, a union is defined in each of the vector classes. This union forces both the __m128 member and the following structure to occupy the same area of memory within the vector class. By doing this, the value of a single element of the __m128 type can be altered by using the member fields X, Y, Z, or W.

Since the union allows access to each individual element of the __m128 data type, it seemed superfluous to include explicit Get() and Set() accessor methods for each field. The use of public fields allowed for a more natural approach to dealing with vector types. However, to ensure true encapsulation and good object-oriented practices are maintained, care has been taken to pass vector classes by const reference whenever possible. The same can be said for the VS_Output and Vertex structures used within the Rasterizer. Passing by const reference ensures that code does not accidentally alter the contents of the object.

Memory Alignment

For Intel SSE instructions to function correctly, the memory location of the data types used must be aligned on a 16 byte boundary. This is due to restrictions within the hardware. The __m128 data type itself contains compiler directives to ensure local instances created on the stack are properly aligned. However, no such protection is given on dynamically allocated instances. As a result, all classes that contain a Vector object value must ensure the object itself is properly aligned.

This can be partially achieved through the use of the `__declspec(align(16))` compiler directive. When placed before the declaration of a class or structure, this directive instructs the compiler to align instances of the following class or structure on a 16 byte boundary, relative to the start of the parent data type.

Data alignment also needs to be ensured when allocating instances dynamically on the heap. This has been achieved by overriding the “new” and “delete” operators in classes containing vector or matrix data types. The macro `DEFINE_NEW_DELETE_ALIGNED` has been defined in `stdafx.h` to provide general purpose 16 byte aligned new and delete operators.

Data alignment was also an issue when dealing with vector<> arrays of vector or matrix types. The standard template library’s vector<> class uses memory allocators to create and delete instances of the given data type on the heap. However, the default allocators are not compatible with aligned data types. To rectify this issue, the definition of each vector<> class involving aligned data types was altered to include a custom memory allocator (5).

Furthermore, Microsoft’s implementation of the std::vector<> type is not fully compatible with aligned data types. There is an issue caused by the resize() method, where the _Val argument is passed by value, not by const reference (6). This causes a compiler error when using aligned data types, as the alignment of an argument passed on the stack cannot be guaranteed. This issue has been corrected by including a modified copy of the `vector.h` header file to replace Microsoft’s. The header has a modified signature for the resize() method, to take the _Val argument by const reference.

Demo Application

The demo application has been written to extensively demonstrate the features of the graphics library. The application cycles through a series of “tests” which show a particular feature of the graphics library. Throughout all tests, the text on the screen displays information about the current rendering mode, number and type of lights, dimensions of the back buffer, and rendering speed in frames per second. The tests are cycled every 15 seconds.

Although the demo application was required to be fully automated, user control support has been included. These controls were used during the debugging phase of development to test the rendering process under various conditions, and were left in the final executable for reference.

The camera within the demo application is controlled under two modes, “Demo Cam” and “Free”. In “Demo Cam” mode, the camera position is dictated by the current demonstration. This is either an orbiting camera, or a fixed camera view point. However, the option is available to switch to “Free” camera mode, which allows the user to control the position of the camera. Note that free camera movement is relative to the current camera direction.

By default, the demo application cycles through each test every 15 seconds. This can be overridden by pausing the demo timer, and manually cycling through each test.

The demo application supports both keyboard input and Xbox 360 controller input using the XInput library. The following table summarises the controls for both input methods:

Function	Keyboard Control	Xbox 360 Control
Toggle Free or Demo Camera	C	Y
Pause Demo Timer	P	Start
Cycle to Next Test	Right Arrow	DPad Right
Cycle to Previous Test	Left Arrow	DPad Left
Close the Application	Escape	Back
Free Camera Controls:		
Forwards	W	Left Stick Up
Backwards	S	Left Stick Down
Left	A	Left Stick Left
Right	D	Left Stick Right
Upward	Q	Right Trigger
Downwards	E	Left Trigger
Rotate Left	Shift + A	Right Stick Left
Rotate Right	Shift + D	Right Stick Right
Rotate Up	Shift + W	Right Stick Up
Rotate Down	Shift + S	Right Stick Down

Works Cited

1. **Microsoft.** Rasterization Rules (Direct3D 9). *Microsoft MSDN Documentation.* [Online] 9 6 2011. [Cited: 9 12 2011.] http://msdn.microsoft.com/en-us/library/windows/desktop/bb147314(v=vs.85).aspx.

2. **Wikipedia.** Bézier curve. *Wikipedia.* [Online] 4 12 2011. [Cited: 9 12 2011.] http://en.wikipedia.org/wiki/B%C3%A9zier_curve.

3. —. Bilinear texture mapping. *Wikipedia.* [Online] 16 11 2011. [Cited: 9 12 2011.] http://en.wikipedia.org/wiki/Bilinear_texture_mapping.

4. **Microsoft.** MMX, SSE, and SSE2 Intrinsics. *Microsoft MSDN Documentation.* [Online] [Cited: 9 12 2011.] http://msdn.microsoft.com/en-us/library/y0dh78ez(v=vs.80).aspx.

5. **blancoding.** AlignedContainerAllocator.h. *code.google.com.* [Online] 28 2 2010. [Cited: 9 12 2011.] http://code.google.com/p/blademaster/source/browse/trunk/BladeMaster/Core/AlignedContainerAllocator.h?r=67.

6. **Ofek.** std::vector of Aligned Elements. *Wordpress.* [Online] 5 5 2010. [Cited: 9 12 2011.] http://thetweaker.wordpress.com/2010/05/05/stdvector-of-aligned-elements/.

7. **Wikipedia.** Barycentric coordinate system (mathematics). [Online] 16 11 2011. [Cited: 9 12 2011.] http://en.wikipedia.org/wiki/Barycentric_coordinate_system_(mathematics).