The ITB Journal

Volume 5 | Issue 2 Article 2

2004

Interactive 2D and 3D Graphics over the World-Wide Web with ActionScript

Hugh McCabe

Recommended Citation

McCabe, Hugh (2004) "Interactive 2D and 3D Graphics over the World-Wide Web with ActionScript," *The ITB Journal*: Vol. 5: Iss. 2, Article 2.
doi:10.21427/D7HX7T
Available at: https://arrow.tudublin.ie/itbj/vol5/iss2/2

Interactive 2D and 3D Graphics over the World-Wide Web with ActionScript

Hugh McCabe

School of Informatics and Engineering

Institute of Technology Blanchardstown

hugh.mccabe@itb.ie

Abstract

Many efforts have been made to provide a mechanism for delivering interactive 3D content over the World-Wide Web. The majority of the solutions put forward require the use of a proprietary plug-in on the client browser and none of these required plug-ins have so far become part of the standard browser installation package. The Macromedia Flash player has however achieved near ubiquitous status as the standard plug-in for displaying interactive multimedia and graphics content. Flash does not provide native support for 3D graphics but the recent addition of the Shape Drawing API opens up the possibility of developers using ActionScript for this purpose. This paper describes the development of an ActionScript graphics library that consists of a set of reusable classes for adding interactive 2D and 3D graphics to Flash content. We describe these classes in detail, show examples of programs developed using them, and discuss the advantages of this approach

1 Introduction

The World-Wide Web (WWW) was initially developed as a means of providing hyperlinked textual information over wide area networks but quickly evolved into a medium that was capable of handle numerous forms of non-textual multimedia content as well; such as bitmapped images, video, and audio [2]. These are typically handled by encoding them into external files adhering to standardised formats whose goal is to ensure efficient coding and playback [11]. In the case of bitmapped images, the browser itself is capable of decoding and displaying the content, and hence the images can be directly embedded in web pages. In the case of video and audio, an external application is often invoked by the browser to handle the playback. It is increasingly common that this external application is developed as a *plug-in* for the browser. In this case seamless embedding of the content within the web page is made possible.

The nature of these content forms mean they are essentially *non-interactive*, in that user input at most comprises of choosing whether to view or play the content or not. *Interactive graphics* on the other hand is a form of multimedia content that is by definition fully interactive [8][9]. In this case the visual content is stored internally as a set of mathematical or quasi-mathematical representations[1] and the display software (or *renderer*) is responsible for generating images for display. A wide variety of interaction is now made possible as mouse and keyboard input can be used to manipulate the internal representations of the objects that make up the scene and the renderer can, in real time, display the results to the user. Interactive graphics is the technology that drives such diverse applications as games [3], computer animation [19], computer-aided design [10], architectural walkthroughs [1], and scientific visualisation [15].

For various reasons, this form of media has not made the same inroads into the World-Wide web as its non-interactive counterparts. The hypertext interface upon which the web is based is unsuited for the sorts of precise mouse and keyboard interactions required, and therefore most attempts to introduce it have revolved around the use of external plug-ins to view and interact with the content. This has been reasonably successful in the case of 2D graphics. The Macromedia Flash system for example [13], provides an effective way of delivering a form of 2D graphics over the web and the associated viewer plug-in, the Flash player, has achieved near total penetration of the browser market[2]. The various ways of delivering 3D graphics however still rely on the presence of plug-ins that the average user is extremely unlikely to have installed in their browser.

In this paper we describe how the Macromedia Flash system can be extended to handle 3D graphics as well as the 2D graphics that it is intended for. The motivations for doing this are as follows:

- Macromedia Flash is a near ubiquitous means of adding multimedia content to the web and its player is installed on the vast majority of the world's web browsers.
- The majority of web-based multimedia developers use Flash and hence they are more likely to experiment with 3D functionality if it is presented in the context of an application they are already familiar with.

[1] As opposed to bitmapped images which comprise of a two-dimensional array of pixel colour values and contain no direct representation of the content.

[2] Macromedia report that an independent survey by NPD research carried out in June 2003 found that 97.4% of web browsers had a version of the Flash player installed.

- Interactive Flash applications are built using the ActionScript language which is relatively simple to learn. From an educational point of view, this work provides an opportunity to get students to experiment with building 3D applications without having to learn complicated high-level graphics languages such as OpenGL or Java3D

This extension is made possible by building a graphics library in ActionScript that uses the Shape Drawing API to render objects on the screen in real time [6]. The rest of this paper consists of the following. Section 2 reviews the methods of delivering interactive graphics over the web and pays particular attention to those that aim to handle 3D graphics. Section 3 describes the Shape Drawing API upon which our library is built and the overall architecture of our system. Section 4 describes the extensions to the Shape Drawing API that our library includes in order to provide a full interactive 2D graphics system. Section 5 details the classes that constitute the 3D part of the graphics library. Section 6 evaluates the system by presenting some sample programs and tests. We finish in Section 7 by outlining what we see as the advantages of this approach and potential applications.

2 Web Graphics Technologies

We now examine what is currently available in terms of technologies for delivering interactive graphics over the web. We will look at technologies for 2D graphics first and then examine the options for 3D.

2.1 2D Graphics

The most important technologies that exist for delivering 2D graphics on the web are Macromedia Flash and Scalable Vector Graphics (SVG) [7]. Flash is a proprietary system developed by Macromedia that has enjoyed widespread adoption across the web. It is a vector graphics system that includes capabilities for doing animation and interactivity. The core element of most Flash content (or *movies* as the Flash system calls them) is the *movie clip*. This is a static rectangular bitmap which can be created by using drawing tools within Flash or by importing graphics or images[3]. These movie clips exist within a 2D coordinate system and can be animated or scripted to behave interactively based on mouse or keyboard input from the user. The scripting language which can be used to control the interactivity is called ActionScript and it provides a rich set of tools for building 2D interactive applications. These applications can be exported in Macromedia's proprietary SWF format and then either embedded in web pages or else ran as standalone applications across the web. Flash has proved ideal for delivering animated content on the web and easily lends itself to more complex tasks such as implementing web-based games.

Scalable Vector Graphics (SVG) [7] has been designed as a non-proprietary alternative to Flash. It effectively defines a standardised XML format for graphical content. There are numerous advantages to encoding graphical content as XML rather than a binary format such as Macromedia SWF. The most important is that it allows developers to freely build applications to create or display this content and therefore could facilitate the development of graphics over the web in the same manner that HTML did for hypertext. SVG is a rich fully featured system but so far plug-ins to view the content have not become nearly as ubiquitous as those for the Macromedia SWF format.

2.2 3D Graphics

There are many options available for the 3D graphics case. Broadly speaking these fall into three categories: systems which convert 3D content into a 2D format such as Flash or SVG; systems which require a custom plug-in or application to view the content; and systems that are built on the basis of using Java applets embedded in web pages. We will look at each of these in turn.

A renderer that is capable of displaying 3D content effectively has to convert this content into a 2D equivalent since ultimately it has to be displayed on a 2D device (the screen). A system such as a 3D game engine has to do this approximately 40 times per second as the state of the world it is rendering is continually updated in real time. However if we are producing a piece of computer animation then there is no such speed requirement as the conversion can be carried out offline. There is no reason why we cannot create a 3D animation in some appropriate application , use another application to convert this into a sequence of 2D bitmaps or 2D vector graphics, and then use a third application to display the results.

[3] The concept of a movie clip is not unlike that of a *sprite*, which was in common usage among early 2D arcade game developers [3].

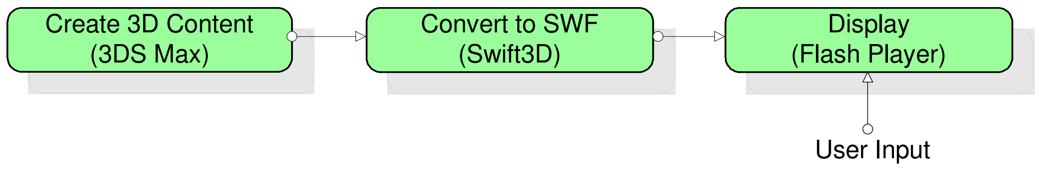


***Figure 1**: The Swift3D Workflow*

This is exactly what systems such as Electric Rain's Swift3D do [5]. This allows the developer to create 3D animated content in an application such as 3DS Max [4] and then use Swift3D to convert that content into the Flash SWF file format (see Figure 1). This means it is viewable on any browser with the Flash player installed. This produces excellent results but the drawback is that since the player can only access the 2D version of the original 3D content no meaningful interaction is possible between the user and that content. Therefore it effectively produces a *non-interactive* form of 3D graphics. Numerous other products exist that work on the same basis and suffer from the same drawbacks.

The second approach involves systems that require a specialised application to view the 3D content. Because this application is built specifically for this purpose it can be designed as a 3D renderer and hence provide full interaction with the 3D content at the user end. This application can then be designed as stand-alone, in which case it is invoked by the browser when content of this form is encountered, or else it can work as a plug-in inside the browser. An early example of this approach is the Virtual Reality Modelling Language (VRML) [20]. This is a tag-based ASCII file format for describing 3D worlds. VRML content can be created by editing by hand or else by using some sort of authoring tool[4]. VRML is an open standard and hence several companies have produced VRML browsers [16] which can be invoked to view the content at the user end. Since the browser the user employs to view the graphics has direct access to the 3D content a rich array of functionality can be provided allowing users to explore 3D worlds and interact with and manipulate the objects therein (Figure 2).

[4] In fact most of the major 3D modelling software packages such as 3DS Max incorporate the functionality to export content as VRML.

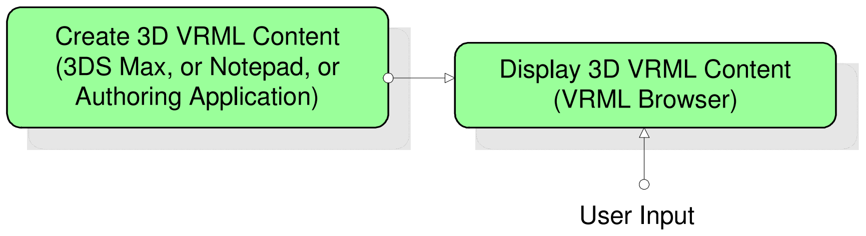

Create 3D VRML Content (3DS Max, or Notepad, or Authoring Application)
Display 3D VRML Content (VRML Browser)
User Input

Figure 2*: The VRML Process*

Despite its technical advantages, VRML (and indeed its successor X3D [21]) has not become a widespread means of delivering 3D graphics mainly due once again to the fact that the browsers needed to access the content have not achieved sufficient market penetration. Another disadvantage of the VRML approach is that the developer has relatively little control over how the eventual viewer can interact with the content. The forms of interaction allowed come pre-packaged as it were with the browser.

Macromedia's Director software is a high-end Multimedia authoring environment that includes support for developing 3D content [12]. With the help of the Shockwave plug-in this content can be viewed and interacted with through a web browser also. Crucially the developer has full control over what form these interactions will take as the powerful Lingo scripting language can be used to specify this. Unsurprisingly the usual problem remains that the necessary Shockwave plug-in does not come as a standard component with web browsers.

The third category of web-based 3D graphics is based on the idea of using a Java applet to display the graphics content. This is a powerful idea as Java includes a 3D graphics library called Java3D [18] that can be used to program more or less any form of 3D graphics interaction we might wish for and modern web browsers are capable of handling Java applets easily. Programming graphics in Java3D is a difficult business however and requires a thorough knowledge of high-level programming concepts. It seems unlikely that multimedia practitioners or students will be willing to invest the time necessary to master these concepts. An interesting alternative is to build an application that provides some sort of straightforward graphical or programmatic interface that allows the user to create the 3D content, and then translates it into a Java 3D program for display in a web browser. An example of this is the Processing language [17] which comprises of a simple and intuitive scripting language whose output is converted into Java applets. This allows the creation of 3D objects through a set of simple classes that have much in common with the work described in this paper.

3 Graphics Library Architecture

Our graphics library is structured as shown in Figure 3.

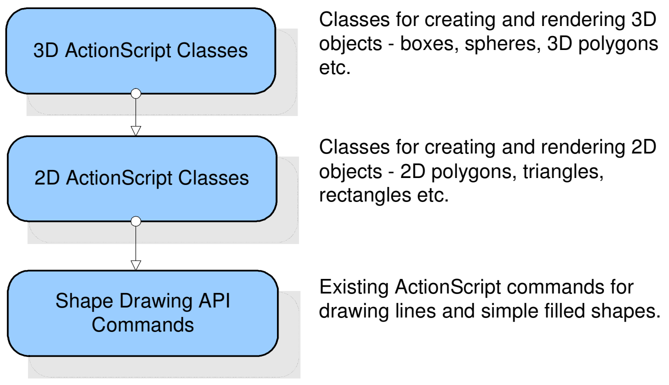

3D ActionScript Classes
Classes for creating and rendering 3D objects - boxes, spheres, 3D polygons etc.
2D ActionScript Classes
Classes for creating and rendering 2D objects - 2D polygons, triangles, rectangles etc.
Shape Drawing API Commands
Existing ActionScript commands for drawing lines and simple filled shapes.

***Figure 3**: Structure of the Graphics Library*

The system is built upon the existing ActionScript Shape Drawing API commands. These were introduced in the most recent version of ActionScript (ActionScript 2.0) and represent a significant development of the language's capabilities. As mentioned earlier Flash graphics are based on the idea of manipulating predefined bitmapped shapes called movie clips. The Shape Drawing API however allows the developer to programmatically draw shapes on the screen dynamically. The commands provided in the Shape Drawing API are quite limited and are summarised in Figure 4.

`linestyle(thickness,rgb,alpha)`	Set the thickness, colour and transparency for subsequent drawing commands
`moveTo(X,Y)`	Move the current drawing position to the point (X,Y)
`lineTo(X,Y)`	Draw a line from the current drawing position to the point (X,Y)
`beginFill(rgb,alpha)`	Set the fill colour and alpha value and begin a fill
`endFill()`	Finish the fill

***Figure 4**: ActionScript Shape Drawing API Commands*

Effectively these commands allow the developer to do two things:

- Draw lines on the screen of varying thicknesses, colours and transparencies.
- Draw filled shapes on the screen of varying colours and transparencies

Lines are drawn using the `moveTo` and `lineTo` commands. Filled shapes are drawn using `beginFill` to set the fill properties, a sequence of line commands to draw a shape that encloses some space on the 2D plane, and then the `endFill` command to invoke the fill. These commands can be issued from ActionScript scripts and hence combined with all of the usual programmatic elements such as conditional statements and loops and also carried out in response to user mouse or keyboard input.

As it stands the Shape Drawing API can carry out sophisticated work but this requires a large amount of programming on the part of the developer. The first task in building our graphics library was to use the basic Shape Drawing commands to create a set of reusable classes for drawing 2D shapes on the screen (see Figure 3). This extends the functionality of the API greatly and allows developers to quickly write code to draw and manipulate 2D graphics. The second stage was to use the 2D drawing classes as a means of implementing a set of 3D classes. We now describe these two stages in more detail.

4 2D ActionScript Classes

There are three basic 2D classes in the library, each of which corresponds to a fundamental graphics primitive. These are `Point2D, Triangle2D,` and `Polygon2D`.

4.1 The `Point2D` class

Objects are defined in terms of their vertices and hence we need a data structure suitable for representing points on a 2D plane. The class is described in Figure 5.

Point2D
X: number Y: number
Point2D(X:number,Y:number):void examine():void drawline(point:Point2D,thickness:number, colour:string, alpha:number):void translate(tx:number,ty:number):void scale(sx:number,sy:number):void rotate(angle:number):void

***Figure 5**: Class diagram for* `Point2D`

The attributes of a `Point2D` object are `X` and `Y` coordinates. Various methods have been implemented including the normal transformations (translation, rotation and scaling), a constructor method, and an `examine` method which prints out the values of the attributes on the screen and is primarily intended for debugging purposes. The `drawline()` method is the fundamental drawing operation of the graphics library and when invoked on a `Point2D` instance, draws a line from this to another `Point2D` instance supplied as an argument. Line thickness, colour and alpha value can also be specified. Colour is represented as a hexadecimal string in the graphics library. This may seem an unintuitive choice to graphics programmers but it is the form that web and multimedia developers are likely to be most familiar with.

```
#include "graphics.as"

p1 = new Point2D();
p2 = new Point2D();
width = _root.width // width of screen/window
height = _root.height // height of screen/window

for(i=0;i<500;i++)
{
  p1.x = Math.random()* width -  (width/2);
  p1.y = Math.random()* height - (height/2);
  p2.x = Math.random()* width -  (width/2);
  p2.y = Math.random()* height - (height/2);
  col = convert(Math.random()*255,Math.random()*255,Math.random()*255);
  thickness = Math.random()*5;
  alpha = Math.random()*100;
  p1.drawline(p2,thickness,col,alpha);
}
```

Figure 6*: Example program using the `Point2D` class*

Figure 6 shows an example ActionScript program that uses this class[5]. The results of running the program are shown in Figure 7.

[5] `Math.random` is the ActionScript method for generating random numbers and the `convert` function converts three integer colour component values into a single hexadecimal colour string. The class definitions are contained in the file `graphics.as`.

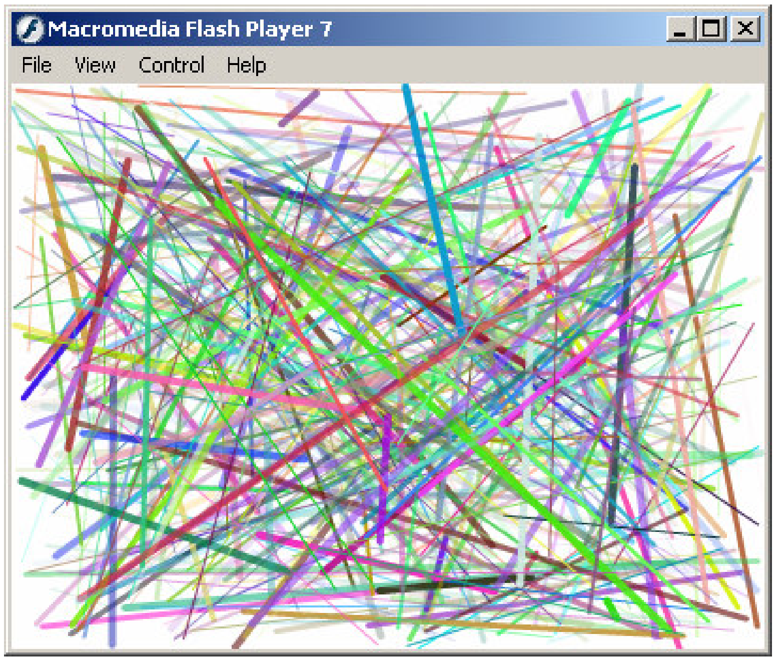


***Figure 7**: Results of running program in Figure 6.*

4.2 The `Triangle2D` Class

Now that we have defined points within our two-dimensional space we can construct a class to represent triangles (see Figure 8).

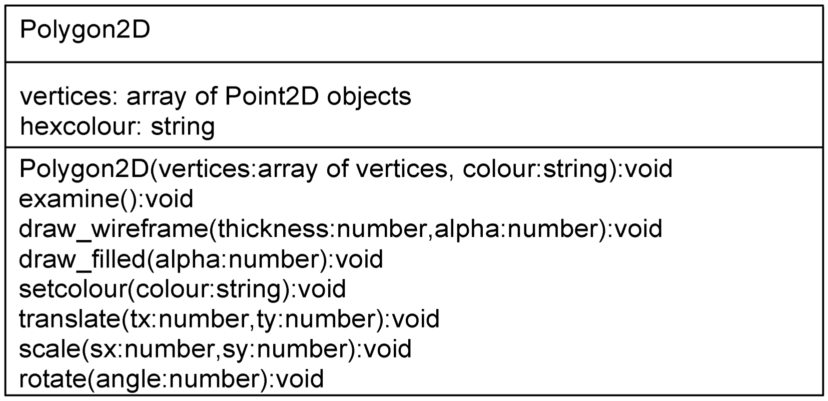
Triangle2D
V1: Point2D V2: Point2D V3: Point2D hexcolour: string
Triangle2D(v1:Point2D, v2:Point2D, v3:Point2D, colour:string):void examine():void draw_wireframe(thickness:number,alpha:number):void draw_filled(alpha:number):void setcolour(colour:string):void translate(tx:number,ty:number) scale(sx:number,sy:number):void rotate(angle:number):void

***Figure 8**: The `Triangle2D` Class*

The `Triangle2D` class has four attributes. Three of these represent the vertices of the triangle and each is an instance of the previously defined `Point2D` class. This means that methods to carry out transformations can be easily implemented by calling the transformation methods of the `Point2D` class. The fourth attribute represents the colour of the triangle. This can be reset at any time using the `setcolour` method. Two drawing methods are provided. One of them draws a wire-frame triangle and the other draws a filled triangle, using the colour of the triangle as the fill colour. The fill is implemented using the ActionScript fill commands described earlier.

4.3 The `Polygon2D` class

The final 2D class provides a way of specifying and manipulating two-dimensional polygons and is shown in Figure 9. This class allows for polygons of an arbitrary number of vertices. The vertices of the polygon are represented as `Point2D` objects and the usual drawing and transformation methods have been implemented.



Polygon2D

vertices: array of Point2D objects
hexcolour: string

Polygon2D(vertices:array of vertices, colour:string):void
examine():void
draw_wireframe(thickness:number,alpha:number):void
draw_filled(alpha:number):void
setcolour(colour:string):void
translate(tx:number,ty:number):void
scale(sx:number,sy:number):void
rotate(angle:number):void

***Figure 9**: The `Polygon2D` class*

The polygon class is extremely similar to the triangle class with the only significant difference being that since a polygon has an arbitrary number of vertices we store the vertex information in an array[6].

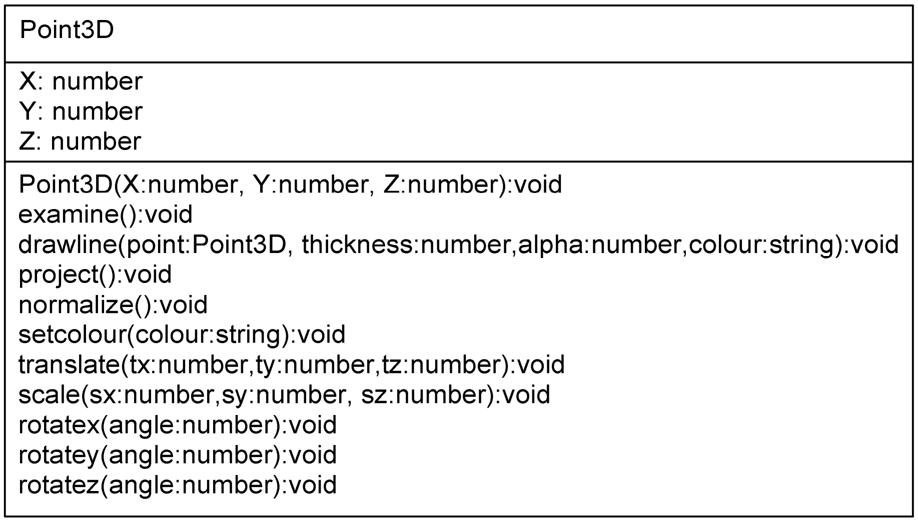
5 3D ActionScript Classes

There are a number of classes to support 3D objects in the library. Obviously the mechanics of implementing these classes are somewhat more complex as we have to take care of issues such as projection from 3D to 2D, and lighting and shading.

5.1 The `Point3D` Class

Once again the basic unit in three-dimensional space is the point and hence we construct a class to support this. The `Point3D` class is shown in Figure 10.

[6] It is worth pointing out that in ActionScript an array is not a fixed size data structure and hence has more in common with what would be called *dynamic array* in a high-level programming language.



```
Point3D
-------------------------------------------------------------------------------
X: number
Y: number
Z: number
-------------------------------------------------------------------------------
Point3D(X:number, Y:number, Z:number):void
examine():void
drawline(point:Point3D, thickness:number,alpha:number,colour:string):void
project():void
normalize():void
setcolour(colour:string):void
translate(tx:number,ty:number,tz:number):void
scale(sx:number,sy:number, sz:number):void
rotatex(angle:number):void
rotatey(angle:number):void
rotatez(angle:number):void
```

Figure 10*: The `Point3D` Class*

Points in 3D space are represented as three numbers. There is a `drawline` method used for drawing a line from one point to another. Note that the implementation of this `drawline` method has to include a projection operation in order to convert from 3D coordinate space into a 2D coordinate system for screen display. The projection provided is the standard perspective projection parallel to the Z-axis of the 3D coordinate system [8]. The specifics of the projection are controlled by two parameters to a `setupcamera` function which would usually be invoked at the start of the program. These parameters are the position of the centre of projection along the Z axis and the distance from the centre of projection to the projection plane. These parameters can be used to simulate the full range of focal lengths and fields-of-view for the projection. A `project` method is also supplied which directly projects the point according to this projection. Since we are dealing with three-dimensional space we provide three rotation methods, for rotating about the three principal axes.

5.2 The Vector3D Class

A vector class is shown in Figure 11. This class contains the methods one would expect for carrying out vector arithmetic. A similar class, `Vector2D`, exists for doing 2D vector arithmetic but is omitted for brevity's sake.

Vector3D
X: number Y: number Z: number
Vector3D(X:number, Y:number, Z:number):void examine():void addition(othervector:Vector3D): result:Vector3D getlength():result:number normalize():void dotproduct(othervector:Vector3D):result:number crossproduct(othervector:Vector3D):result:Vector3D

***Figure 11**: The `Vector3D` class*

5.3 The `Triangle3D` Class

3D triangles are at the heart of the graphics library and we support the rendering of both wire-frame and shaded triangles. Given the existence of a `Triangle2D` class the algorithms for rendering 3D triangles are quite simple. The process involves projecting the vertices of the 3D triangle and forming a 2D triangle from the results. This 2D triangle can then be drawn on the screen using the methods from the `Triangle2D` class.

We also need to take care of the problem of hidden surface removal. This involves detecting those triangles that are not visible to the viewer and declining to render them accordingly. Our library currently supports backface removal [9][8] which uses vector arithmetic to determine if a triangle is facing away from the viewer. If the triangle is facing away from the viewer then, as long as the object is convex, it must be at the back and therefore hidden. The algorithm for rendering a wire-frame 3D triangle is shown in Figure 12.

```
Algorithm: Rendering a wire-frame triangle

calculate normal vector to triangle
if (normal vector.view vector < 0)
      calculate projected vertices
      draw 2D triangle from projected vertices
```

***Figure 12**: Algorithm for rendering wire-frame triangle*

The library also supports the rendering of filled 3D triangles. We use the *constant shading* [8] algorithm to calculate colour values for the projected triangles. To support this, the library allows the user to introduce an arbitrary number of point light sources into the scene. For each light source we add a colour component to the polygon which is calculated as proportional to the strength of the light source and the cosine of the angle between the surface normal and the vector towards the light source. Effectively this technique calculates diffuse reflection from the object.

```
Algorithm: Rendering a filled triangle

calculate normal vector to triangle
set colour to (0,0,0)
if (normal vector.view vector < 0)
      calculate projected vertices
      for each light source
            calculate vector to light source
            calculate cosine of angle
            add cosine.lightsourcecolour to colour
      add ambient light term to colour
      draw 2D coloured triangle from projected vertices
```

***Figure 13**: Rendering a Filled Triangle*

The complete class diagram for the Triangle3D class is shown in Figure 14.

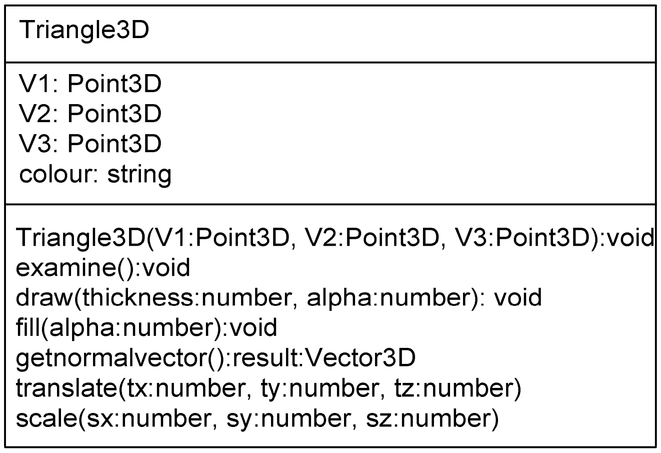

Triangle3D
V1: Point3D
V2: Point3D
V3: Point3D
colour: string
Triangle3D(V1:Point3D, V2:Point3D, V3:Point3D):void
examine():void
draw(thickness:number, alpha:number): void
fill(alpha:number):void
getnormalvector():result:Vector3D
translate(tx:number, ty:number, tz:number)
scale(sx:number, sy:number, sz:number)

***Figure 14**: Class diagram for* `Triangle3D`

5.4 The `Polygon3D` Class

Once again the triangle is basically a specialisation of the polygon. A polygon contains an arbitrary number of vertices and these vertices are held in an array in our implementation.

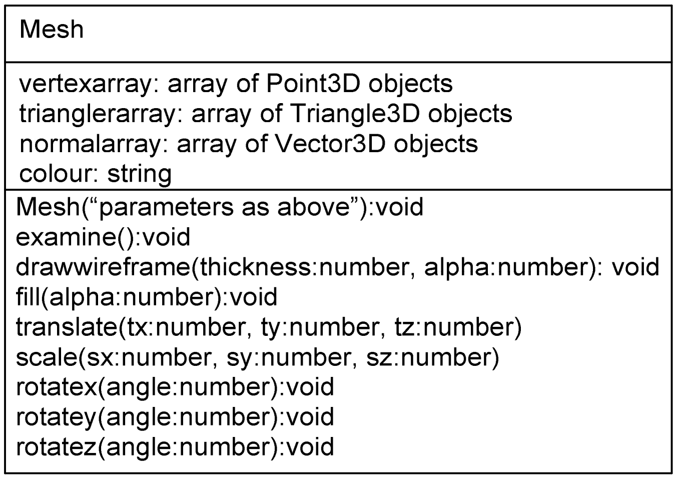
The same methods are provided as for the `Triangle3D` class thus allowing the rendering of both wire-frame and filled polygons.

5.5 The `Mesh` Class

For convenience sake it is necessary to introduce a higher level object representation method. Our library accomplishes this by providing a mesh class. The `Mesh` class is designed to handle polygonal meshes. These are collections of polygons that approximate the surfaces of 3D objects. A mesh can be represented as an array of polygons but it is more normal to insist that these polygons are restricted to being triangles. The attributes of a mesh object are:

- An array of 3D triangles
- An array of vertices
- An array of normal vectors
- A colour attribute

The array of vertices consists of a non-repeating list of the vertices of the triangles in the triangle array and can be derived from this. This is provided to speed up the rendering process. We also store an array of normal vectors to the triangles in order to avoid having to recalculate them each time we want to do rendering or lighting calculations with them. The class diagram is shown in Figure 15.



```
Mesh

vertexarray: array of Point3D objects
trianglerarray: array of Triangle3D objects
normalarray: array of Vector3D objects
colour: string

Mesh("parameters as above"):void
examine():void
drawwireframe(thickness:number, alpha:number): void
fill(alpha:number):void
translate(tx:number, ty:number, tz:number)
scale(sx:number, sy:number, sz:number)
rotatex(angle:number):void
rotatey(angle:number):void
rotatez(angle:number):void
```

***Figure 15**: Class diagram for the `Mesh` class*

To go with the `Mesh` class a number of functions are provided in the library for generating meshes. These include functions for primitive shapes such as spheres and octagons, and also functions to import vertex information from 3DS Max models.

6 Test Programs

We now briefly present some results of test programs that demonstrate the use of the various classes in the graphics library. The first program uses the `Triangle3D` class to create twelve 3D triangles arranged in the shape of a box. The fill method is used to render the triangles and therefore constant shading is employed. Some ActionScript code was written to respond to key presses to that the user can rotate the box on the screen around the three principal axes. A screenshot is shown in Figure 16.

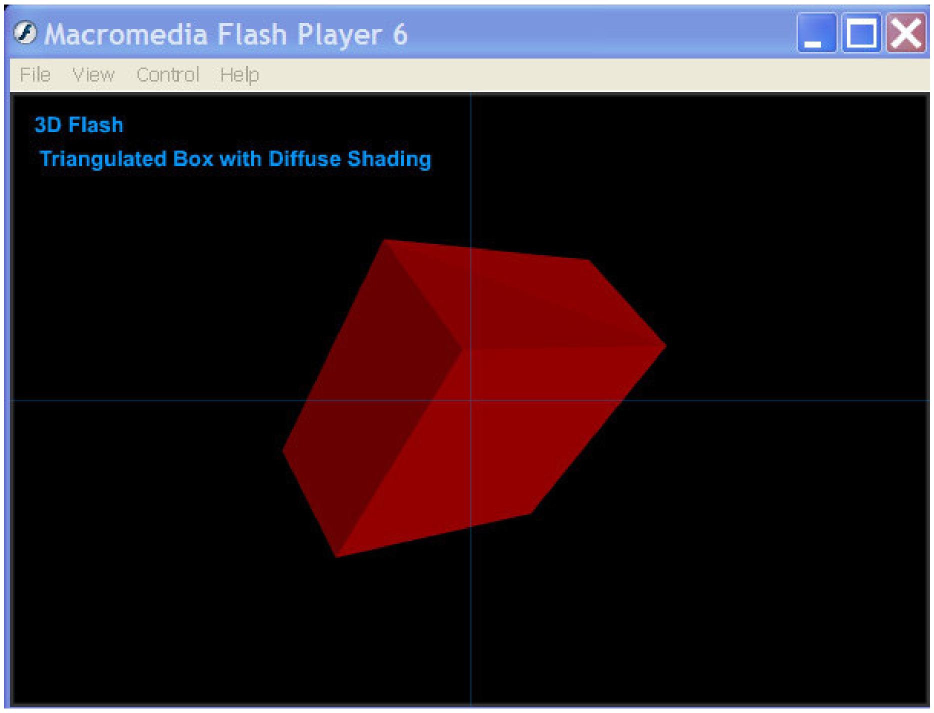

Macromedia Flash Player 6
File View Control Help
3D Flash
Triangulated Box with Diffuse Shading

***Figure 16**: 3D Box created from the `Triangle3D` class*

As can be seen the constant shading technique proves an effective method in this case. The rendering is carried out quickly and smooth motion results when the rotations are applied.. It is also possible to create a box shape from the `Polygon3D` class. In this case only six polygons would be necessary as opposed to the twelve triangles used above.

The next program serves as a demonstration of the `Mesh` class. A mesh is constructed which represents a diamond shape and is shown in. Figure 17.

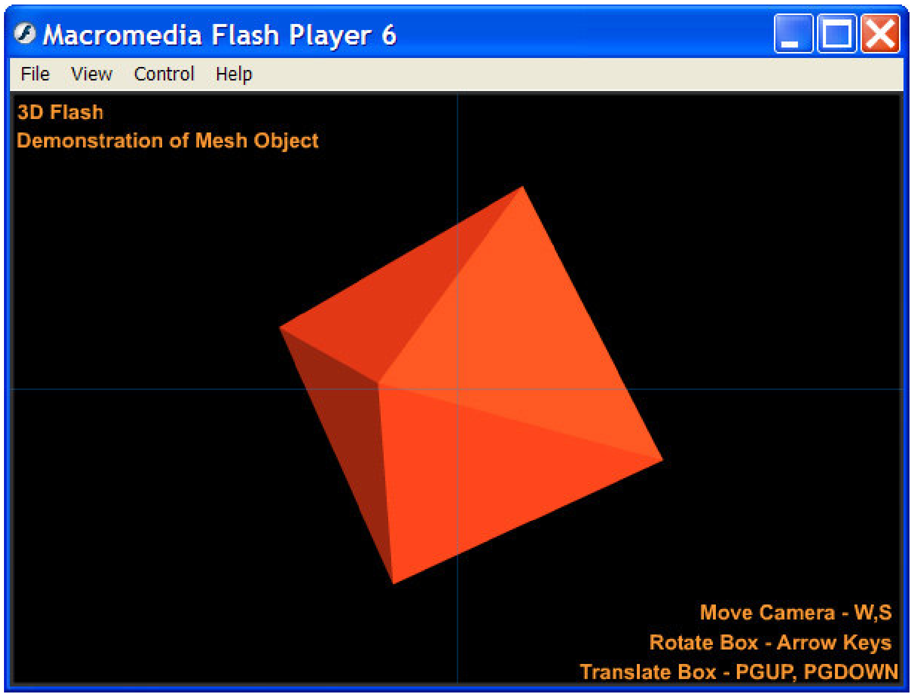

Macromedia Flash Player 6
File View Control Help
3D Flash
Demonstration of Mesh Object
Move Camera - W,S
Rotate Box - Arrow Keys
Translate Box - PGUP, PGDOWN

***Figure 17**: Using the `Mesh` class*

Once again a set of keyboard controls are provided which allow the user to rotate the object but also translate the object and move the camera. High frame rates and smooth motion were found to be possible in this example also.

A final example was constructed in order to test the efficiency of the rendering system when using larger numbers of polygons. In order to facilitate this a library function was written which generates a mesh approximating the surface of a sphere. The input parameters to this algorithm include the radius of the sphere and the level of recursion that the algorithm should use. A larger number of recursion levels results in a finer polygonal mesh and a greater number of polygons. The results of rendering such a spherical mesh object are shown in Figure 18. The faceted nature of the sphere is caused by the use of the constant shading algorithm which makes polygon edges visible. We discuss this issue further in the conclusions section to this paper.

As would be expected the higher number of triangles involved here places extra demands on the processor. A series of tests were carried out by instructing the sphere generation algorithm to generate spheres of increasing polygon counts[7]. The results of this are shown in Figure 19.

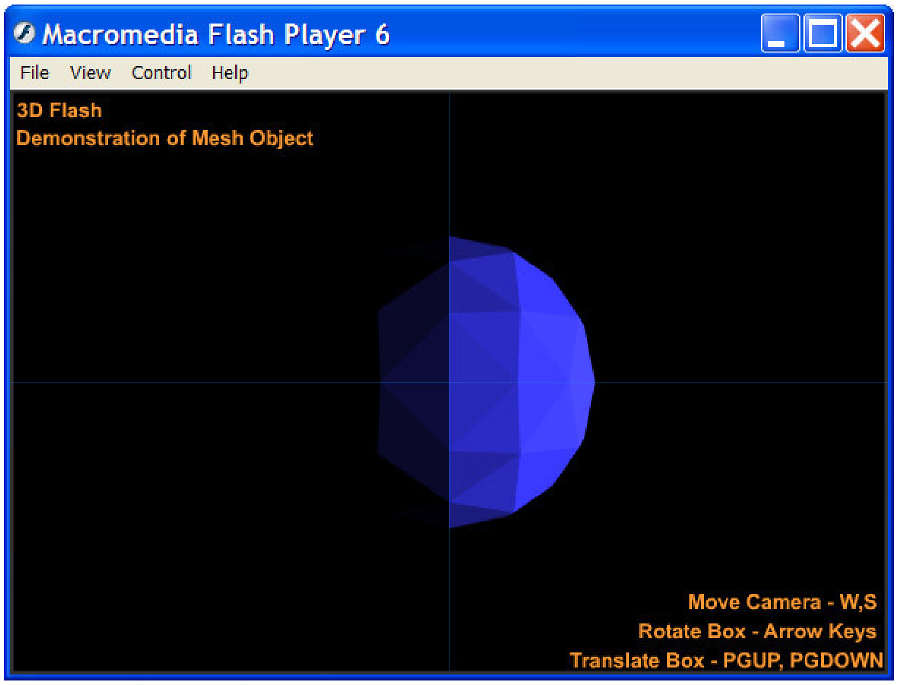

Macromedia Flash Player 6
File View Control Help
3D Flash
Demonstration of Mesh Object
Move Camera - W,S
Rotate Box - Arrow Keys
Translate Box - PGUP, PGDOWN

Figure 18*:The sphere mesh*

Number of Poylgons	Acceptable Frame Rate
8	Yes
32	Yes
128	No
512	No

Figure 19*: Results of sphere generation test*

An acceptable frame rate was deemed to be a situation where the program responded immediately to key presses to rotate the sphere and was capable of producing smooth motion. The 128 polygon case only fell slightly short of these criteria and hence the exact number of polygons that the system can handle is closer to this upper limit than the lower one of 32.

[7] These tests were carried out on a laptop computer with a Pentium-M 1.5 GHz processor, 512MB of RAM and a 64MB graphics card.

7 Conclusions

We have demonstrated that it is possible to use the ActionScript Shape Drawing API as a basis for developing a graphics library that Flash developers can use to easily program interactive 2D and 3D graphics. We presented a set of classes that encapsulate the fundamental 2D and 3D graphics primitives and contain appropriate methods for rendering and manipulating these primitives. We conclude by discussing the advantages and disadvantages of this approach and suggesting some avenues for future work in this area.

From the point of view of programming 2D graphics, the library is extremely successful. The functionality of the Shape Drawing API is hugely increased as the developer can now easily add shapes such as triangles and polygons, and also make use of the various transformation methods to manipulate these shapes. Perhaps more importantly, it grounds the 2D graphics elements within the well understood framework of standard coordinate geometry for computer graphics, and this results in a far more powerful and flexible approach than the rather ad hoc methods of doing things which Flash developers typically employ.

The 3D part of the library also provides greatly extended functionality as at present ActionScript offers no 3D functionality at all. The library allows the developer to create various forms of 3D shapes, including a polygonal mesh. These shapes can be rendered in either wire-frame or shaded form, and the full set of transformations for manipulating these shapes interactively within the 3D space are also provided. Some limitations of this aspect of the work should be addressed however.

- **Polygon Count**. As we saw earlier polygonal objects with three figure polygon counts prove too prohibitive in terms of computational expense to render in shaded form. This is unsurprising as the nature of the Flash system means that programs written with ActionScript are unable at present to take advantage of the hardware graphics acceleration that is necessary to carry out tasks such as this. It is likely that careful optimisation of the graphics library will improve this situation somewhat.

- **Hidden Surface Removal**. The backface removal algorithm which has been implemented has the advantage of being extremely fast but it does suffer from obvious and well-known limitations. These are, that it will not correctly remove hidden surfaces in the situation where multiple objects are obscuring each other, and it will not work correctly with non-convex objects. The nature of the program that the developer is trying to write will dictate whether these limitations are important or not. If, for example, the objects are being rendered in wire-frame mode anyway then it is likely that this will not be important. This problem could be solved by implementing a full hidden surface removal algorithm such as Z-Buffer [8]. This is only really feasible however if we can take advantage of hardware acceleration. Similar problems were grappled with in the graphics research community in the 1970's, before hardware acceleration became the norm, and the algorithms proposed are now being investigated to determine their potential suitability in the present situation.

- **Shading**. Filled objects are shaded using the constant shading technique, with the assumption that surfaces exhibit diffuse reflection only. This works perfectly well for shapes where the polygons represent the actual surfaces of the object, but less well for curved surfaces where the polygons are an *approximation* of the surface of the object. Once again this problem could be easily solved by implementing a more sophisticated algorithm, such as Gourand shading [9]. The computational demands of Gourand are not significantly higher than the present algorithm but in this case we run into a practical difficulty regarding the Shape Drawing API. Gourand shading advises that projected triangles are interpolatively shaded scan-line by scan-line but the fill methods of the Shape Drawing API do not allow for this. A form of Gourand shading could be implemented by shading on a pixel-by-pixel basis but it is likely that the computational demands of this would be so much that the visual advantages would be far outweighed by the speed disadvantages.

It should be noted that most of the issues raised above are only really important if the developer is seeking to implement realistic looking 3D graphics. Realism is not going to be achieved by a graphics library in ActionScript due to the computational limitations and the lack of it should not be regarded as a serious problem.

Perhaps the chief advantage of providing a way of doing this kind of graphics through ActionScript is that the developer can freely mix this form of media with all of the others provided by Flash. The concept of layers within the Flash movie means that on one layer we can have 3D graphics, and can then combine this with multiple other layers containing text, sprites, video, drawings, bitmaps, and so on. These layers can be overlaid and composited together with various levels of transparency perhaps, or used to provide different forms of media on different segments of the screen. Due to the ubiquity of the Flash player the results can be viewed across the World-Wide Web. We believe that creative Flash developers can potentially use this library to create exciting new forms of 2D and 3D graphics applications.

Acknowledgements

The work described in this paper benefited enormously from numerous discussions with Matt Smith on Macromedia Flash and ActionScript. A prototype version of this library was used for teaching graphics to the National Diploma in Computing class of 2003/2004 at the Institute of Technology Blanchardstown. The comments and input of the students proved extremely valuable.

References

[1] Aliaga, D.G., and Lastra, A.A. *Architectural Walkthroughs using Portal Textures*. In Proceedings of 8th IEEE Visualization '97 Conference, Phoenix, AZ, 1997.

[2] Berners Lee, T., Cailliau, R., Luotonen, A., Nielsen, H. F., and Secret, A. *The World-Wide Web*. Communications of the ACM archive Volume 37, Issue 8 (August 1994), pages 76-82, 1994.

[3] DeMaria, R., and Wilson, J.L. *High Score! The Illustrated History of Electronic Games*. 2nd Edition, McGraw-Hill Osborne Media, 2003.

[4] Discreet, Autodesk, Inc. 3DS Max. http://www.discreet.com/products/3dsmax

[5] Electric Rain. http://www.erain.com

[6] Ewing, R. *Introduction to Macromedia Flash MX Drawing Methods*. Available at http://www.macromedia.com/devnet/mx/flash/articles/precision_drawing.html

[7] Ferraiolo, J. *Scalable Vector Graphics (SVG) 1.0 Specification,* 4 September 2001. Available at http://www.w3.org/TR/SVG. 2001.

[8] Foley, J.D., van Dam, A., Feiner, S.K., and Hughes, J.F. *Computer Graphics, Principles and Practice*. 2nd Edition in C, The Systems Programming Series, Addison-Wesley, 1996.

[9] Hearn, D., and Baker, M.P. *Computer Graphics with OpenGL.* 3rd Edition, Prentice-Hall, 2003.

[10] Hoschek, J. *Fundamentals of Computer Aided Geometric Design*. A.K. Peters, Ltd., 1993.

[11] Le Gall, D. *MPEG: A Video Compression Standard for Multimedia Applications*. Communications of the ACM archive, Volume 34, Issue 4 (April 1991), pages 46-58, 1991.

[12] Macromedia. http://www.macromedia.com

[13] Macromedia, Inc. Flash Software Documentation. Available at http://www.macromedia.com/flash., 2004.

[14] Macromedia, Inc. Flash Player for Developers and Publishers. Available at http://www.macromedia.com/software/flash/survey/whitepaper_jul03.pdf, 2003

[15] Nielson, G.M., Shriver, B.D., and Rosenblum, L.J. *Visualization in Scientific Computing*. Los Alamitos, Calif., IEEE Computer Society Press, 1990.

[16] Parallel Graphics. http://www.parallelgraphics.com

[17] Processing. http://www.processing.org

[18] Sun Micorsystems, Inc. Java 3D 1.1 API Specification. Mountain View, CA, December 1998. Available at http://www.javasoft.com

[19] Thalman, N.M. *Computer Animation: Theory and Practice*. Springer-Verlag New York, 1985.

[20] The VRML Consortium. VRML International Standard ISO/IEC 14772-1:1997. Available at http://www.vrml.org

[21] The Web3D Consortium. http://www.web3d.org

[22] Wallace, G. K. *The JPEG Still Picture Compression Standard*. Communications of the ACM archive, Volume 34, Issue 4, (April 1991), pages 30-44, 1991