Blotch3D User Manual

[There are a few "TBD"s *in the below text indicating that a feature is not fully implemented or there is no example of it]*

Introduction

Blotch3D is a C# library that vastly simplifies development of 3D applications and games.

Examples are provided that show how with just a few lines of code you can…

1. Load standard file types of 3D models (as "sprites"), textures, fonts, etc. and display and move them in 3D with real-time performance.
2. Create dynamic sprites (custom model vertices).
3. Attach sprites to other sprites to create sprite trees as deep as you want. Child sprite orientation and position is relative to its parent sprite's orientation and position, and can be changed dynamically. (It's a scene graph.)
4. Override all steps in the drawing of each sprite.
5. A default GUI allows the user to control camera position, orientation, zoom, etc.
6. Programmatically control camera position, orientation, zoom, etc.
7. Create billboard sprites.
8. Create imposter sprites [TBD].
9. Show in-world and 2D text in any font, size, and color at any 2D or 3D position, including a sprite's position.
10. Connect sprites to the camera to implement HUD objects and text that resizes with the window size, etc.
11. Connect the camera to a sprite to implement 'cockpit view'.
12. Implement GUI controls in the 3D window.
13. Implement a skybox.
14. Get a list of sprites touching a ray, to implement weapons fire, etc.
15. Get a list of sprites under the mouse position, to implement mouse selection, tooltips, pop-up menus, etc.
16. Detect collisions between sprites [example TBD].
17. Implement levels-of-detail.
18. Implement mipmaps.
19. Implement translucent sprites.
20. Support stereoscopic views (anaglyph, VR, etc.) [TBD].
21. Implement fog [example TBD].
22. Use with WPF and WinForms.
23. Under Microsoft Windows, access and override many window features and functions using the provided WinForms Form object of the window.
24. Build for other platforms (currently supports iOS, Android, MacOS, Linux, all Windows platforms, PS4, PSVita, Xbox One, and Switch).

Blotch3D uses MonoGame. MonoGame is a widely used 3D library for C# (see Wikipedia article on the professional games based on it). It is free, fast, cross platform, actively developed by a large community, and fully implements Microsoft's (no longer supported) XNA4 engine. There is a plethora of MonoGame/XNA4 documentation, tutorials, examples, and discussions on line. All MonoGame features remain available in Blotch3D.

All reference documentation of Blotch3D (classes, methods, fields, properties, etc.) is available through Visual Studio IntelliSense. If you are using another IDE that doesn't support IntelliSense, just look at the comment directly in the Blotch3D source. If you aren't getting useful IntelliSense information for a keyword, it may be a MonoGame keyword rather than a Blotch3D keyword. In that case you can look it up online.

Quick start

Get the MonoGame installer from http://www.monogame.net/downloads/ and run it.
Get the Blotch3D repository zip from https://github.com/Blotch3D/Blotch3D and unzip it.
Open the Visual Studio solution file.
Build and run the example projects. They are each comprised of a single small source file demonstrating one aspect of Blotch3D.

Developing with Blotch3D

The provided solution contains both the Blotch3D library project with source, and the example projects.

BlotchExample01_Basic is a bare-bones Blotch3D application, where GameExample.cs contains the example code. Other example projects also contain a GameExample.cs, which is the same source file from BlotchExample01_Basic but with a few additions to it to demonstrate the feature of the example. In fact, you can do a diff between the BlotchExample01_Basic source file and another example's source file to see what extra code must be added to implement the features it demonstrates [TBD].

All provided projects build only for the Windows platform. To create a new project for Windows you can just copy the BlotchExample01_Basic folder and rename the project, or you can create the project from scratch like this:

1. Select File/New/Project and create a 'MonoGame Windows Project'.
2. Select 'Build/Configuration Manager' and create the platform you want (like x64) and check the build box.
3. Open the project properties and specify '.NET Framework 4.6'.
4. Select output type of 'Console Application' for now, so you can see any debug messages. You might want to change this back to 'Windows Application' later.
5. Add a reference to the Blotch3DWindows assembly.
6. Rename the Game1 file and class as desired.
7. Open that file.
8. Add a 'using Blotch' line at the top of the file.
9. Have the class inherit from BlWindow3D instead of "Game", delete its body, and add overrides of Setup, FrameProc, and/or FrameDraw as desired.

To create a project for another platform (Android, iOS, etc.), make sure you have the Visual Studio add-on that supports it (for example, for Android you'll need to add the Xamarin Android feature), and follow something like the above steps for that platform. Or if that doesn't work, look online for an example.

All model meshes, textures, fonts, etc. used by the 3D hardware must be created and accessed by the same thread, because supported hardware platforms require it (like OpenGL etc.). Its best to assume all Blotch3D and MonoGame objects should be created and accessed in that thread.

When you instantiate a class derived from BlWindow3D, it will create the 3D window and make it visible, and create a single thread that we'll call the "3D thread". (This pattern is used because MonoGame uses it. In fact, the BlWindow3D class inherits from MonoGame's "Game" class. But instead of overriding Initialize, LoadContent, Update, and Draw, you override Setup, FrameProc, and FrameDraw from BlWindow3D. Other "Game" class methods and events can still be overridden, however.)

Although it may apparently work in certain circumstances, do not have the class constructor create or access any 3D resources, or have its instance initializers do it, because neither are executed by the 3D thread. Initialization of 3D resources should be done in the Setup method.

Code to be executed in the context of the 3D thread must be in the Setup, FrameProc, and/or FrameDraw methods, because those methods are automatically called by the 3D thread. Also, see below for information on how another thread can queue a delegate to the 3D thread. A single-threaded application does everything in those overridden methods, as follows:

When you override the Setup method it will be called once when the object derived from BlWindow3D is first created. You might put time-consuming initialization of persistent things in there like graphics setting initializations if different from the defaults, loading of persistent content (models, fonts, etc.), creation of persistent BlSprites, etc. Do not draw things in the 3D window from the setup method.

When you override the FrameProc method it will be called once per frame (you control frame period with BlGraphicsDeviceManager.FramePeriod). You can put code there that should be called periodically. This is typically code that must run at a constant rate, like code that implements smooth sprite and camera movement, etc. For a single-threaded application, this is also where you put your application code. Do not draw things in the 3D window from the FrameProc method.

When you override the FrameDraw method, the 3D thread calls PrepareDraw just before calling FrameDraw once per frame, but more rarely if CPU is being exhausted. This is where you put drawing code (BlSprite.Draw, BlGraphicsDeviceManager.DrawText, etc.). For more efficiency in a single-threaded application that may exhaust the CPU, you can put the periodic code here that should be executed periodically, instead of in FrameProc. But then it should adjust itself to account for variations in period.

If you are developing a multithreaded app, then when other threads need to create, change, or destroy 3D resources or otherwise do something in a thread-safe way with the 3D thread, they can pass a delegate to EnqueueCommand or EnqueueCommandBlocking. Those methods make sure the code is done by the 3D thread sequentially at the end of the next FrameProc call. If anything needs to be conveyed back to app threads, you can create thread-safe queues for that as well. For example, user input to the 3D window may need to be conveyed to other threads in a multi-threaded application.

Most Blotch3D objects must be Disposed when you are done with them. You can check the IsDisposed member to see if an object has been disposed.

See the examples and use IntelliSense for more information.

Making 3D models

There are several primitive models available with Blotch3D. The easiest way to add them to your project is to…

1. Copy the Content folder from the Blotch3D project folder to your project folder
2. Add the "Content.mgcb" file in that folder to your project
3. Right-click it and select "Properties"
4. Set the "Build Action" to "MonoGameContentReference"

You can get the names of the content files that you can use by starting the pipeline manager (double-click Content/Content.mgcb). You can also add more content via the pipeline manager (see http://rbwhitaker.wikidot.com/monogame-managing-content). See the examples for details on how to load and display models, fonts, etc.

If no existing model meets your needs, you can either programmatically create a model by specifying the vertices (see the custom Vertices example), or create a model with the Blender 3D modeler. You can also instruct Blender to include texture (UV) mapping, by watching one of the countless tutorials online, like https://www.youtube.com/watch?v=2xTzJIaKQFY or https://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/UV_Map_Basics .

Dynamically changing a sprite's orientation and position

Each sprite has a "Matrix" object that defines its orientation and position relative to its parent sprite. When you change a sprite's orientation and position, you also change the orientation and position of its child sprites. So the sprites "connected" to a sprite (the subsprites) will follow that sprite's orientation, position, scale, shear, etc.

You can do a lot with Blotch3D/MonoGame without knowing anything about the internal workings of a matrix object. Mainly you only need to know that…

1. A matrix is an object that describes a coordinate system relative to a parent's coordinate system.
2. There are many static and instance methods of the Matrix class that let you create matrices for scaling, translation, rotation, etc.
3. There are also static and instance Matrix methods and operator overloads to combine (matrix multiply) matrices to form a single matrix which combines the effects of multiple matrices. For example, a rotate matrix and a scale matrix can be multiplied to form a single rotate-scale matrix. But mind the multiplication order. To combine matrices, you would multiply them in the reverse order you would apply them in real life. For example, if conceptually you want to translate (move) and then rotate an object, multiply the rotation matrix by the translate matrix rather than the translate matrix by the rotation matrix. Novices can simply try the operation one way and, if it doesn't work the way you wanted, do it the other way.

For a really good introduction (without the math), see http://rbwhitaker.wikidot.com/monogame-basic-matrices. (That site has many great MonoGame tutorials.)

The rest of this section should be studied only when you need a deeper knowledge.

Internals of the Matrix

Here we'll introduce the internals of 2D matrices. 3D matrices simply have one more dimension.

First, a few of definitions:

1. A "coordinate system" is a set of points whose position is defined relative to each other.
2. The "origin" of a coordinate system is the point we define as the "starting point" for defining other points. For example, another point might be defined as being 3 to the right and 5 up from the origin, notated by (3,5).
3. Often, we use the words "point" and "vertex" (plural "vertices") interchangeably. But more specifically a "vertex" is a point in the coordinate system that is used for something. For example, it may be the corner of a model.

A point on a plane can be described with a horizontal distance from the origin (the point's "X" value) and a vertical distance from the origin (the point's "Y" value), notated by (X,Y).

For example, our model might have one vertex 4 to the right and 1 up from the origin, notated by (4,1), and another vertex 3 to the right and 3 up from the origin, notated by (3,3). (This is a very simple model comprised of only two vertices.)

You can move the model by moving each of those vertices by some amount without regard to how far they currently are from the origin. To do that, just add an offset vector to each vertex. For example, we could add the vector (2,1) to each of those original vertices, which would result in final vertices of (6,2) and (5,4). In that case we have *translated* (moved) the model.

Matrices support translation, but first let's talk about moving a vertex *relative to its current position from the origin,* because that's what gives matrices the power to shear, rotate, and scale a model. This is because those operations affect each vertex differently depending on its relationship to the origin.

If we want to change the X of each vertex from its current horizontal distance from the origin by a factor of 2, we can multiply the X of each vertex by 2. For example,

$X' = 2X$ (where X' is the final value)

… which would change the above vertices from (4,1) and (3,3) to (8,1) and (6,3). In that case we have *scaled* the model relative to the origin (in this case only in the X direction).

We might want to define how to change each X according to the original X value of each vertex *and also according to the original Y value*, like this:

$X' = aX + bY$

For example, if a=0 and b=1, then this would set the new X of each vertex to its original Y value.

Finally, we might also want to define how to create a new Y according to the original X and original Y. So, the equations for both the new X and new Y are:

$X' = aX + bY$
$Y' = cX + dY$

Remember, the idea is to apply this to every vertex. By convention we might write the four numbers in a 2x2 matrix, like this:

a b
c d

This should all be very easy to understand.

But why are we even talking about it? Because now we can define the values of a matrix that, if applied to each vertex of a model, define any type of *transform* in the position or orientation of that model.

For example, if we apply the following matrix to each of the object's vertices:

1 0
0 1

…then the vertices are unchanged, because…

X' = 1X + 0Y
Y' = 0X + 1Y

…sets X' to X and Y' to Y.

This matrix is called the *identity* matrix because the output is the same as the input.

We can create matrices that scale, shear, and even rotate points. To make an object three times as large, use the matrix:

3 0
0 3

To scale only X by 3 (stretch an object in the X direction), then use the matrix:

3 0
0 1

The following matrix flips (mirrors) the model vertically:

1 0
0 -1

Below is a matrix to rotate an object counterclockwise by 90 degrees:

0 -1
1 0

A matrix can be created to rotate any amount about any axis.

(The Matrix class provides functions that make it easy to create a rotation matrix from a rotation axis and angle, or pitch and yaw and roll, or something called a quaternion, since otherwise we'd have to call sine and cosine functions, ourselves, to create the matrix elements.)

Since we often also want to translate (move) points *without* regard to their current distances from the origin as we did at the beginning of this section, we add more numbers to the matrix just for that purpose. And since many mathematical operations on matrices work only if the matrix has the same number of rows as columns, we add more elements simply to make the rows and columns the same size. And since Blotch3D/MonoGame works in 3-space, we add even more numbers to handle the Z dimension. So, the final matrix size in 3D graphics is 4x4.

Specifically:

X' = aX + bY + cZ + d
Y' = eX + fY + gZ + h
Z' = iX + jY + kZ + l
W = mX + nY + oZ + p

(Consider the W as unused, for now.)

Which can be notated as…

a	b	c	d
e	f	g	h
I	j	k	l
m	n	o	p

Notice that the d, h, and l are the translation vector.

The Matrix class in MonoGame uses the following field names:

M11	M12	M13	M14
M21	M22	M23	M24
M31	M32	M33	M34
M41	M42	M43	M44

Besides the ability to multiply entire matrices (as mentioned at the beginning of this section), you can also divide (i.e. multiply by a matrix inverse) matrices to, for example, solve for a matrix that was used in a previous matrix multiply, or otherwise isolate one operation from another. Welcome to linear algebra! We won't get in to how matrix multiplication and division specifically process the individual elements of the matrix because the Matrix class already provides those functions.

As was previously mentioned, each sprite has a matrix describing how that sprite and its children are transformed from the parent sprite's coordinate system. Specifically, Blotch3D does a matrix-multiply of the parent's matrix by the child's matrix to create the final matrix used to draw that child, and it is also used as the parent matrix for the subsprites of that sprite.

Because of confusion in coordinate system handedness (chirality), multiplication/division order, row vs. column notation (mathematicians use the opposite notation of that used by 3D graphics people), and the order of element storage in memory; on occasion it may be easier to try things one way and, if it doesn't work as expected, try it another way. But for details see http://seanmiddleditch.com/matrices-handedness-pre-and-post-multiplication-row-vs-column-major-and-notations.

A Short Glossary of 3D Graphics Terms

Vertex
A point in space. Typically, a point at which the line segments of a polygon meet. That is, a corner of a polygon. A corner of an object. Most visible objects are described as a set of vertices. Each vertex can have a color, texture coordinate, and normal.

Polygon
A visible surface described by a set of vertices that define its corners. A triangle is a polygon with three vertices, a quad is a polygon with four. One side of a polygon is a "face".

Ambient lighting
A 3D scene has one ambient light setting. The intensity of ambient lighting on the surface of a polygon is unrelated to the orientation of the polygon or the camera.

Diffuse lighting
Directional or point source lighting. You can have multiple directional or point light sources. Its intensity depends on the orientation of the polygon relative to the light.

Texture
A 2D image applied to the surface of an object. For this to work, each vertex of the object must have a texture coordinate associated with it, which is an X,Y coordinate of the 2D bitmap image that should be aligned with that vertex. Pixels across the surface of a polygon are interpolated from the texture coordinates specified for each vertex.

Normal
In mathematics, the word "normal" means a vector that is perpendicular to a surface. In 3D graphics, "normal" means a vector that indicates from what direction light will cause a surface to be brightest. Normally they would mean the same thing. However, by defining a normal at some angle other than perpendicular, you can somewhat cause the illusion that a surface lies at a different angle. Each vertex of a polygon has a normal vector associated with it and the brightness across the surface of a polygon is interpolated from the normals of its vertices. So, a single flat polygon can have a gradient of brightness across it giving the illusion of curvature. In this way an object composed of fewer polygons can still be made to look quite smooth.

X-axis
The axis that extends right from the origin.

Y-axis
The axis that extends forward from the origin.

Z-axis
The axis that extends up from the origin.

Translation
Movement. The placing of something at a different location from its original location.

Rotation

The circular movement of each vertex of an object about the same axis.

Scale
A change in the width, height, and/or depth of an object.

Shear (skew)
A pulling of one side of an object in one direction, and the opposite side in the opposite direction, without rotation, such that the object is distorted rather than rotated. A parallelogram is a rectangle that has experienced shear. If you apply another shear along an orthogonal axis of the first shear, you rotate the object.

Yaw
Rotation about the Y-axis

Pitch
Rotation about the X-axis, after any Yaw has been applied.

Roll
Rotation about the Z-axis, after any Pitch has been applied.

Euler angles
The yaw, pitch, and roll of an object, applied in that order.

Matrix
An array of 16 numbers that describes the position and orientation of a sprite. Specifically, a matrix describes a difference, or transform, in the orientation (coordinate system) of one object from another. See Introduction to Matrices.

Origin
The center of a coordinate system. The point in the coordinate system that is, by definition, at (0,0).

Frame
In this document, 'Frame' means a complete still scene. It is analogous to a movie frame. A moving 3D scene is created by drawing successive frames—typically at about 15 to 60 times per second.

Features and limitations

1. You are welcome to write multithreaded code in your 3D application. However, any code that directly accesses 3D hardware resources (textures, vertices, etc.) must be executed by the BlWindow3D's 3D thread (see Developing with Blotch3D for details). This is because of a limitation in some of the underlying graphics subsystems (OpenGL, etc.). Since it's hard to know what parts of the library access hardware, one should assume all accesses of Blotch3D objects should be done in the 3D thread.
2. At the time of this writing, MonoGame was not designed with a goal of supporting multiple 3D windows because many platforms it supports are not conducive to it. Even if you close the first window before opening the second, the second window won't work right. (You *can* create them, but they don't work correctly and in certain situations will crash.) If you want to be able to "close" and "re-open" a window, you can just hide and show the same window. (On Microsoft Windows,

you can use the WinForms BlWindow3D.Form object for that.) Support for multiple windows may be added to MonoGame in the future.

3. To make the MonoGame window be a child window of an existing GUI, you need to explicitly size, position, and convey Z order. On Microsoft Windows, the window's Form object (BlWindow3D.Form) may be of help in this.
4. MonoGame fully implements Microsoft's (no longer supported) XNA 4 engine, but for multiple platforms. It also implements features beyond XNA 4. So understand that XNA 4 documentation you come across may not show you the best way to do something, and documentation of earlier versions of XNA (versions 2 and 3) will often not be correct. For conversion of XNA3 to XNA4 see http://www.nelsonhurst.com/xna-3-1-to-xna-4-0-cheatsheet/.

Rights