Blotch3D User Manual

Create 3D apps starting with a few lines of example code

Quick start

(This quick start section is for Windows. See below for other platforms like Android, etc.)

1. Get the installer for the latest release of MonoGame from http://www.monogame.net/downloads/ and run it. (Do NOT get the current development version nor the NuGet package.)
2. Get the Blotch3D repository zip from https://github.com/Blotch3D/Blotch3D and unzip it.
3. Open the Visual Studio solution file.
4. Build and run the example projects. They are each comprised of a single small source file demonstrating one aspect of Blotch3D [TBD: elaborate examples need to be split into simpler examples].
5. See IntelliSense comments for reference documentation that (surprisingly) describes more than the obvious.

Introduction

Blotch3D is a C# library that vastly simplifies many of the fundamental tasks in development of 3D applications and games.

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

- Load standard file types of 3D models as "sprites" and display and move them in 3D with real-time performance.
- Set a model's material, texture, and how it responds to lighting.
- Load textures from standard image files.
- Show 2D and in-world (as a texture) text in any font, size, color, etc. at any 2D or 3D position, and make text follow a sprite in 2D or 3D.

- Attach sprites to other sprites to create associated structures of multiple sprites as large as you want. Child sprite orientation and position is relative to its parent sprite's orientation and position, and can be changed dynamically. (Sprite trees are dynamic scene graphs.)
- Override all steps in the drawing of each sprite.
- You can give the user easy control over all aspects of the camera (zoom, pan, truck, dolly, rotate, etc.).
- Easily control all aspects of the camera programmatically.
- Create billboard sprites.
- Create imposter sprites [TBD].
- Connect sprites to the camera to implement HUD models and text.
- Connect the camera to a sprite to implement 'cockpit view', etc.
- Implement GUI controls (as dynamic 2D text or image rectangles) in the 3D window.
- Implement a skybox.
- Get a list of sprites touching a ray, to implement weapons fire, etc.
- Get a list of sprites under the mouse position, to implement mouse selection, tooltips, pop-up menus, etc.
- Detect collisions between sprites [TBD: no example yet].
- Implement levels-of-detail.
- Implement mipmaps.
- Implement translucent sprites and textures with an alpha channel.
- Support stereoscopic views (anaglyph, VR, etc.) [TBD].
- Implement fog [TBD: no example yet].
- Create sprite models programmatically (custom vertices).
- Use with WPF and WinForms.
- Access and override many window features and functions using the provided WinForms Form object of the window (Microsoft Windows only).
- Build for many platforms (currently supports iOS, Android, MacOS, Linux, all Windows platforms, PS4, PSVita, Xbox One, and Switch).

Blotch3D sits on top of MonoGame. MonoGame is a widely used 3D library for C#. It is free, fast, cross platform, actively developed by a large community, and it's used in many professional games. There is a plethora of MonoGame documentation, tutorials, examples, and discussions on line. All MonoGame features remain available. For example, custom shaders can be written to override the default shader.

All reference documentation of Blotch3D (classes, methods, fields, properties, etc.) is available through Visual Studio IntelliSense. It explains how and why you use the feature, and answers frequent questions. 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 need to look it up online.

See MonoGame.net for the official MonoGame documentation. When searching on-line for other MonoGame documentation and discussions, be sure to note the MonoGame version being discussed. Documentation of earlier version may not be compatible with the latest.

MonoGame fully implements Microsoft's (no longer supported) XNA 4 engine, but for multiple platforms. It also implements features beyond XNA 4. Therefore 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 XNA 3 to XNA 4 see http://www.nelsonhurst.com/xna-3-1-to-xna-4-0-cheatsheet/.

Note that to support all the platforms, certain limitations were necessary. Currently you can only have one 3D window. Also, there is no official cross-platform way to specify an existing window to use as the 3D window—MonoGame must create it. See below for details and work-arounds.

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 similar to the one from the basic example but with a few additions to it to demonstrate a certain feature. 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: the "full" example needs to be split to several simpler examples].

All provided projects are configured to build only for the Windows platform. To create a new project for Windows you can just copy the basic example 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.cs file and class as desired.
7. Replace its contents with code from an example.

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 Xamarin Android), and follow something like the above steps for that platform, or look online for instructions on creating a MonoGame project for that platform.

If you are copying the Blotch3D assembly (like Blotch3D.DLL on Windows) to a project or packages folder so you don't have to include its source code in a project, be sure to also copy Blotch3D.XML so you still get the IntelliSense comments. You shouldn't have to copy any other binary file from the Blotch3D output folder if you've installed MonoGame on the destination machine. Otherwise you should copy the entire output folder. For example, you'd probably want to copy everything in the Blotch3D output folder when you are distributing your app.

You must 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", which calls certain methods of the derived class.

This pattern is used because MonoGame uses it. In fact, the BlWindow3D class inherits from MonoGame's "Game" class. But instead of overriding MonoGame's Initialize, LoadContent, Update, and Draw, you override Blotch3D's Setup, FrameProc, and FrameDraw from BlWindow3D. Other "Game" class methods and events can still be overridden, if needed.

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

Code to be executed in the context of the 3D thread must be in the Setup, FrameProc, and/or FrameDraw methods of the class derived from BlWindow3D, because those methods are automatically called by the 3D thread. A single-threaded application would have all its code in those overridden methods. For a multi-threaded application, other threads that need to do 3D things can queue a delegate to the 3D thread as described below.

Although it may apparently work in certain circumstances or on certain platforms, do not have the BlWindow3D-derived class constructor create or access any 3D resources, or have its instance initializers do it, because neither are executed by the 3D thread.

The 3D thread calls the Setup method once at the beginning of instantiation. You might put time-consuming initialization of persistent things in there like loading of persistent content (sprite models, fonts, etc.), creation of persistent BlSprites, etc.

The 3D thread calls the FrameProc method once per frame (you control frame period with BlGraphicsDeviceManager.FramePeriod). For single-threaded applications this is typically where the bulk of application code resides, except the actual drawing code. For multi-threaded applications, this is where all application code resides that does anything with 3D resources.

The 3D thread calls the FrameDraw method every frame, but only if there is enough CPU for the 3D thread. Otherwise it calls it less frequently. This is where you put drawing code (BlSprite.Draw, BlGraphicsDeviceManager.DrawText, etc.). Additionally, if you are developing a single-threaded application (i.e. everything is in the 3D thread) that will also be very subject to exhausting its thread, then you can put the application code in FrameDraw rather than in FrameProc—as long as the code adjusts itself to account for variations in how often it is called. This may save CPU when it is being exhausted because the FrameDraw is called less often in that case.

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.

For multithreaded applications, besides keeping all 3D code in the 3D thread, you must of course follow rules you would for any multithreaded app. Specifically, in a 64-bit app on 64-bit hardware, accessing a reference or primitive data type is naturally thread safe. That is, any single primitive type 64-bits long or less, like a reference (which internally is a pointer), floating point value, integer, etc., is naturally atomic. But any data that must be accessed atomically with multiple steps (like atomic accesses to multiple variables, reading/writing structures, read-modify-writes, or accesses to variables larger than the 64-bit data bus size) must be done by only one thread or passed as a delegate to the same thread (case in point,

the EnqueueCommand or EnqueueCommandBlocking of the 3D thread), or all threads must hold a mutex or use a critical section when accessing that data. If you use a mutex, you must make sure there can be no deadlocks with other mutexes. A critical section blocks all other threads regardless, but can't ever deadlock and has less overhead otherwise.

MonoGame does not support multiple 3D windows because that isn't conducive on certain platforms. On Microsoft Windows (and possibly certain other platforms) 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.)

To make the MonoGame window be a child window of an existing GUI, you need to explicitly size, position, and convey Z order so that it is overlaid at the right screen location. The easiest way to do that would be to overlay the 3D window on an existing child window by getting the current attributes of that child window, whenever they change. On Microsoft Windows, the window's Form object (BlWindow3D.Form) may be of help in this. There may also be a way to specify that an existing window be used as the 3D window, but it probably isn't portable and may not work in later MonoGame releases.

Most Blotch3D objects must be Disposed when you are done with them and you are not otherwise terminating the program.

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 by starting the MonoGame 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, for example, the Blender 3D modeler and then add it to the project with the pipeline manager. 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" member that defines its orientation and position relative to its parent sprite. There are many static and instance methods of the Matrix class that let you easily set and change the scaling, translation, rotation, etc. of a matrix.

When you change anything about a sprite's matrix, you also change the orientation and position of its child sprites, if any. That is, subsprites reside in the parent sprite's coordinate system.

There are also static and instance Matrix methods and operator overloads to combine (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. See below for details, but novices can simply try the operation one way (like A times B) and, if it doesn't work the way you wanted, do it the other way (B times A).

For a good introduction (without the math), see http://rbwhitaker.wikidot.com/monogame-basic-matrices.

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

Matrix internals

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

Let's imagine a model that has one vertex at (4,1) and another vertex at (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 the same amount, and without regard to where each is relative to 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 model vertices of (6,2) and (5,4). In that case we have *translated* (moved) the model.

Matrices certainly 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 about the origin. This is because those operations affect each vertex differently depending on its relationship to the origin.

If we want to scale (stretch) the X relative to the origin, 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).

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 for each vertex according to its 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 matrix elements (a, b, c, and d) 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 elements of a matrix that, if applied to each vertex of a model, define any type of *transform* in the position and orientation of that model.

For example, if we apply the following matrix to each of the model'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 (X',Y') is the same as the input (X,Y).

We can create matrices that scale, shear, and even rotate points. To make a model three times as large (relative to the origin), use the matrix:

3 0
0 3

To scale only X by 3 (stretch a model 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 a model 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 matrices because the Matrix class already provides those static or instance 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 child.

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 a model. Most visible models 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 a model. For this to work, each vertex of the model 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 a model 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 a model about the same axis.

Scale
A change in the width, height, and/or depth of a model.

Shear (skew)
A pulling of one side of a model in one direction, and the opposite side in the opposite direction, without rotation, such that the model 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 model.

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 a model, 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 model from another. See Dynamically changing a sprite's orientation and position.

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.

Rights

WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.