# Blotch3D

# Quick start for Windows

On your development machine …

1. Get the installer for the latest release of MonoGame from <http://www.monogame.net/downloads/> and run it with the default settings. (Do NOT get the current development version nor a NuGet package.)
2. Get the Blotch3D repository zip from <https://github.com/Blotch3D/Blotch3D> and unzip it.
3. Open the Visual Studio solution file (Blotch3D.sln).
4. Build and run the example projects.
5. Use IntelliSense to see the reference documentation, or see “Blotch3DManual.pdf”.
6. To create a new Blotch3D project, or add Blotch3D to an existing project, or build for another platform, follow the instructions in the [Creating a new project](#_Creating_a_new) section.

# Features

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

Bare-bones examples are provided that show how with just a few lines of code you can…

* Load standard 3D model file types as “sprites”, and display and move thousands of them in 3D at high frame rates.
* Set a sprite’s material, texture, and lighting response.
* Load textures from standard image files, including textures with an alpha channel (i.e. with translucent pixels).
* 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 ‘sprite trees’ as large as you want. Child sprite orientation, position, scale, etc. are relative to the parent sprite, and can be changed dynamically (i.e. the sprite trees are dynamic scene graphs.)
* Override all steps in the drawing of each sprite.
* Easily give the user control over all aspects of the camera (zoom, pan, truck, dolly, rotate, etc.).
* Easily control all aspects of the camera programmatically.
* Create billboard sprites.
* Show a video as a 2D or 3D texture (See <http://rbwhitaker.wikidot.com/video-playback> for details).
* 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, and with transparent pixels in the 3D window.
* Implement a skybox sprite.
* Get a list of sprites touching a ray (within a sprite radius), to implement weapons fire, etc.
* Get a list of sprites under the mouse position (within a sprite radius), to implement mouse selection, tooltips, pop-up menus, etc.
* Implement levels-of-detail.
* Implement mipmaps.
* Implement height fields (a surface with a height that maps from an image).
* Implement 3D graphs (a surface with a height that follows an equation or an array of height values).
* Dynamically transform (animate) a texture.
* Create sprite models programmatically (custom vertices).
* Use with WPF and WinForms, on Microsoft Windows.
* Access and override many window features and functions using the provided WinForms Form object of the window (Microsoft Windows only).
* Detect sprite radius collisions.
* Implement fog.
* Define ambient lighting, and up to three point-light sources. (More lights can be defined if a custom shader is used.)
* All other MonoGame feature remain available.
* Build for many platforms. Currently supports all Microsoft Windows platforms, iOS, Android, MacOS, Linux, 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 used in many professional games. There is a plethora of MonoGame documentation, tutorials, examples, and discussions on line.

Reference documentation of Blotch3D (classes, methods, fields, properties, etc.) is available through Visual Studio IntelliSense, and in “Blotch3DManual.pdf”. Note: To support Doxygen, links in the IntelliSense comments are preceded with ‘#’.

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 versions 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/.](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.

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

“BlotchExample01\_Basic” is a bare-bones Blotch3D application, where Example.cs contains the example code. Other example projects also contain an Example.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 basic Examples.cs 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 the provided projects are configured to build for AnyCPU and the Microsoft Windows x64 platform. See below for details on other platforms.

# Creating a new project

To develop with Blotch3D, you must first follow the steps in the [Quick start](#_Quick_start) section to install MonoGame. Then…

To create a new project from scratch, select File/New/Project/MonoGame, and select the type of MonoGame project you want. Then add the source, or a reference to the source, of Blotch3D.

To add MonoGame plus Blotch3D to an existing non-MonoGame project, add a reference to the appropriate MonoGame binary (typically in “\Program Files (x86)\MSBuild\MonoGame\v3.0\...”). Also add a reference to, or the source of, Blotch3D.

To create a project for another platform besides Microsoft Windows, generally you follow the same procedure described here but you will need to install any Visual Studio add-ons, etc. for the desired platform. For example, for Android you’d need the Xamarin for Android add-on. For some platforms you may need to do some research to create a project.

To distribute a program, deliver everything in your project’s output folder.

# Development

See the examples and their comments, starting with the basic example.

3D subsystems (OpenGL, DirectX, etc.) generally require that all 3D hardware resources be accessed by a single thread. MonoGame follows this rule, and thus you must follow the rule in your project. (There are certain platform-specific exceptions, but MonoGame does not use them.)

To make a 3D window, you must derive a class from BlWindow3D and override the Setup, FrameProc, and FrameDraw methods. When it comes time to create the 3D window, you instantiate that class and call its “Run” method *from the same thread that instantiated it*. The Run method will call the Setup, FrameProc, and FrameDraw methods when appropriate (explained below), and not return until the window closes. (For this reason, you may want to create the BlWindow3D from within some other thread than the main thread so that the main thread can handle a GUI or whatever).

We will call the abovementioned thread the “3D thread”.

The rule to access 3D hardware resources by a single thread also applies to any code structure (Parallel, async, etc.) that may internally use other threads, as well. Since sometimes it’s hard to know exactly what 3D task really does hit the 3D hardware, its best to assume all of them do (like creation and use of Blotch3D and MonoGame objects).

The Setup, FrameProc, and FrameDraw methods are called by the 3D thread as follows:

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

The FrameProc method is called by the 3D thread once every frame. 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 typically where all application code resides that does anything with 3D resources, except the actual drawing code. (Note: You can also pass a delegate to the BlSprite constructor, which will cause that delegate to be executed every frame. The effect is the same as putting the code in FrameProc, but it better encapsulates sprite-specific code.)

Once every frame the 3D thread prepares for drawing and then calls the FrameDraw method, but only if there is enough CPU available in the 3D thread. Otherwise FrameDraw is called less frequently. This is where you should put drawing code (BlSprite.Draw, BlGraphicsDeviceManager.DrawText, etc.). For apps that may suffer from severe CPU exhaustion (at least for the 3D thread), you may also want to put your app code in this method, as well, so it is called less frequently, assuming that application code can properly handle being called at variable rates.

You can use a variety of methods to draw things in FrameDraw. Blotch3D provides methods to draw text and textures in 2D (just draw them after all 3D objects have been drawn so they aren’t overwritten by them). Sprites are drawn with the BlSprite.Draw method. When you draw a sprite, all its subsprites are also drawn. So, oftentimes you may want to have a “Top” sprite that holds others, and call the Draw method of the Top sprite to draw all other sprites. (BlSprite inherits from Dictionary<string, BlSprite>, where the string key is the subsprite name.) You can also draw things directly with MonoGame. For example, it is faster to draw multiple 2D textures and text using MonoGame’s SpriteBatch class.

By default, lighting, background color, and sprite coloring are set so that it is most probable you will see them. These may need to be changed after you’ve verified sprites are properly created and positioned.

A single-threaded application would have all its code in the three overridden methods: Setup, FrameProc, and FrameDraw. If you are developing a multithreaded program, then you would probably want to reserve the 3D thread only for tasks that access 3D hardware resources. When other threads do need to create, change, or destroy 3D hardware resources or otherwise do something in a thread-safe way with the 3D thread, they can pass a delegate via BlWindow3D.EnqueueCommand or BlWindow3D.EnqueueCommandBlocking.

Because multiple windows are not conducive to some of the supported platforms, MonoGame, and thus Blotch3D, do not support more than one 3D window. (You can create any number of other, non-3D windows you like.) You can *create* multiple 3D windows, but MonoGame does not support them correctly (input sometimes goes to the wrong window) 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.

Officially, MonoGame must create the 3D window, and does not allow you to specify an existing window to use as the 3D window. There are some platform-specific ways to do it described online, but note that they may not work in later MonoGame releases.

To properly make the MonoGame window be a child window of an existing GUI, you need to explicitly size, position, and convey Z order to the 3D window so that it is overlaid over the child window. The BlWindow3D.WindowForm field will be useful for this (Microsoft Windows only).

All MonoGame features remain available and accessible in Blotch3D. For examples:

* The models you specify for a sprite object (see the BlSprite.LODs field) are MonoGame “Model” objects or a VertexPositionNormalTexture array.
* The BlWindow3D class derives from the MonoGame “Game” class. The Setup, FrameProc, and FrameDraw methods are called by certain overridden Game methods.
* The BlGraphicsDeviceManager class derives from MonoGame’s “GraphicsDeviceManager” class.
* You are welcome to draw MonoGame objects along with Blotch3D objects.
* All other MonoGame features are available, like audio, etc.

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

See the examples, reference documentation (doc/Blotch3DManual.pdf), and IntelliSense for more information.

# Making and using 3D resources

All 3D resources like models, texture images, fonts, etc. must be “compiled” into “XNB” files by MonoGame’s *pipeline manager*. (Although Blotch3D does provide a way to load an image file directly.)

The Blotch3D project already has a pipeline manager and several models that are compiled when Blotch3D is built. If the source to Blotch3D is included in your solution, you can use the provided models (the various resolution spheres, plane, torus, images, etc.) as is shown in the examples without worrying about where the XNB files are. You can also just copy the XNB files from the Blotch3D output folder to a project’s output folder.

To create a new model, you can either programmatically create it by specifying the vertices and normals (see the example that creates custom vertices), or create a model with, for example, the Blender 3D modeler. You can also instruct Blender to include texture (UV) mapping by using 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> .

Once the model is created, it must be compiled to an XNB file by a pipeline manager. (The pipeline manager can import several model file types.) To do this, you can either add that standard model file to the pipeline manager in the Blotch3D project (double-click the Content.mgcb file in the Blotch3D project. See <http://rbwhitaker.wikidot.com/monogame-managing-content> for details.) so that it gets compiled next time Blotch3D is built, or you could make sure the resource is added to a pipeline manager in your own project.

When you create a MonoGame project from scratch, a Content.mgcb file is added to the project by the MonoGame project wizard, and you can start the pipeline manager by double-clicking that file. If you have a project without a Content.mgcb file and you want it to have one, then do the following…

1. Copy the Content folder from the Blotch3D project folder (or any other MonoGame project with a content 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”

If the “MonoGameContentReference” build option is not available in the drop-down (for example, because the project wasn’t originally created by the MonoGame wizard), then try this:

(from <http://www.infinitespace-studios.co.uk/general/monogame-content-pipeline-integration/>)

1. Open your application .csproj in a text Editor.
2. In the first <PropertyGroup> section add <MonoGamePlatform>$(Platform)</MonoGamePlatform>, where $(Platform) is the system you are targeting e.g Windows, iOS, Android. For example: <MonoGamePlatform>Windows</MonoGamePlatform>
3. Add the following lines right underneath the <MonoGamePlatform /> element: <MonoGameInstallDirectory Condition="'$(OS)' != 'Unix' ">$(MSBuildProgramFiles32)</MonoGameInstallDirectory>

<MonoGameInstallDirectory Condition="'$(OS)' == 'Unix' ">$(MSBuildExtensionsPath)</MonoGameInstallDirectory>

1. Find the <Import/> element for the CSharp (or FSharp) targets and underneath add:

<Import Project="$(MSBuildExtensionsPath)\MonoGame\v3.0\MonoGame.Content.Builder.targets" />

# Custom effects

Blotch3D provides several custom shaders that are the same as the default MonoGame BasicEffect, but they provide added features. Examples are provided that demonstrate how to use them.

The shader source and the compiled shader files are in the Blotch3D Content/Effects folder. To use a custom shader, first copy the shader file to your program’s output folder (for example, you could add a link to it in your project and set its properties so it is copied to the output folder).

When your program runs, it specifies that file in the BlBasicEffect constructor. Then when the sprite is drawn, the effect must be “applied”.

Each effect also typically has certain parameters that must be specified that control the feature provided by the effect. These are set with the BlBasicEffect.Parameters[].SetValue method. They can be set at any time.

For example, the BlBasicEffectAlphaTest effect is used like this:

// Create the BlBasicEffect and specify the shader file (you can also specify ‘BlBasicEffectAlphaTestOGL.mgfxo’ if you are on an OpenGL platform)

MyBlBasicEffectAlphaTest = new BlBasicEffect(Graphics.GraphicsDevice, “BlBasicEffectAlphaTest.mgfxo”);

// Now specify the alpha threshold below which pixels are not drawn.

// This can be done at any time, including from within the below delegate

MyBlBasicEffectAlphaTest.Parameters["AlphaTestThreshold"].SetValue(.3f);

// set the custom effect for the sprite

MyTranslucentSprite.SetEffect = (s,effect) =>

{

// Setup the standard BasicEffect texture and lighting parameters

s.SetupBasicEffect(MyBlBasicEffectAlphaTest);

return MyBlBasicEffectAlphaTest;

};

The shader source code (HLSL) for each BlBasicEffect shader is just the original MonoGame BasicEffect shader code with a few lines added. If for some reason you want to recompile the effects, use the “make\_effects.bat” file in the Blotch3D source folder to build them. But first be sure to add the path to 2MGFX.exe to the ‘path’ environment variable. Typically the path is something like “\Program Files (x86)\MSBuild\MonoGame\v3.0\Tools”.

You can create your own shader files that are based on BlBasicEffect and compile and load it as shown above. Just be sure it is based on the original HLSL code for BasicEffect, or one of the provided custom shaders.

# Translucency

Each pixel of a texture has a red, a green, a blue intensity value. Some textures also have an “alpha” value for each pixel, to indicate how translucent the pixel should be. Specifically, the alpha value indicates how much of any coloration behind that pixel (farther from the viewer) should show through the pixel. Alpha values of 1 indicate the texture pixel is opaque and no coloration from farther values should show through. Values of zero indicate the pixel is completely transparent.

Translucent textures drawn using the 2D Blotch3D drawing methods (BlGraphicsDeviceManager#DrawText, BlGraphicsDeviceManager#DrawTexture, and BlGuiControl) or any MonoGame 2D drawing methods (for example, by use of MonoGame’s SpriteBatch class) will always correctly show the things behind them according to the pixel’s alpha channel as long as they are called after all other 3D things are drawn.

But translucent textures applied to a 3D sprite may require special handling.

If you simply apply the translucent texture to a sprite as if it’s just like any other texture, you will not see through the translucent pixels when they happen to be drawn chronologically *before* anything farther away, because drawing the near surface also updates the depth buffer (see Depth Buffer in the glossary). Since the depth buffer records the nearer pixel, it prevents further pixels from being drawn afterward. For some translucent textures the artifacts can be negligible, or your particular application may avoid the artifacts entirely because of camera constraints, sprite position constraints, and drawing order. In those cases, you don’t need any other special code. We do this in the “full” example because the draw order of the translucent sprites, and their positions, are such that you won’t see the artifacts because you can’t even see the sprites when viewed from underneath, which is when you would otherwise see the artifacts in that example. (Note: subsprites are drawn in the order they are added to the parent sprite.)

One way to mitigate most of these artifacts is by using alpha testing. Alpha testing is the process of completely neglecting to draw transparent texture pixels, and thus neglecting to update the depth buffer at that window pixel. Most typical textures with an alpha channel use an alpha value of only zero or one (or close to them), indicating absence or presence of visible pixels. Alpha testing works well with textures like that. For alpha values specifically intended to show partial translucency (alpha values nearer to 0.5), it doesn’t work well. In those cases, you can either live with the artifacts, or beyond that at a minimum you will have to control translucent sprite drawing order (draw all opaque sprites normally, and then draw translucent sprites far to near), which will take care of all artifacts except those that occur when sprites intersect or two surfaces of a single sprite occupy the same screen pixel. For some scenes it might be worth it to draw without updating the depth buffer at all (do a "Graphics.GraphicsDevice.DepthStencilState = Graphics.DepthStencilStateDisabled” in the BlSprite.PreDraw delegate, and set it back to DepthStencilStateEnabled in the BlSprite.DrawCleanup delegate). These are only partial solutions to the alpha problem. You can look online for more advanced solutions.

The default MonoGame “Effect” used to draw models (the “BasicEffect” effect) uses a pixel shader that does not do alpha testing. MonoGame does provide a separate “AlphaTestEffect” effect that supports alpha test. But AlphaTestEffect is *not* based on BasicEffect (and therefore must be handled differently in code), and does not support directional lights, as are supported in BasicEffect. So, don’t bother with AlphaTestEffect unless you don’t care about the directional lights (i.e. you are using only emission lighting). (If you do want to use AlphaTestEffect, see online for details.)

For these reasons Blotch3D includes a custom shader file called BlBasicEffectAlphaTest (to be held in code as a BlBasicEffect object) that provides everything that MonoGame’s BasicEffect provides, but also provides alpha testing. Set its “AlphaTestThreshold” to specify what alpha value merits drawing the pixel. See the [Custom effects](#_Custom_effects) section and the SpriteAlphaTexture example for details.

# Dynamically creating an alpha channel

Blotch3D includes a BlBasicEffectClipColor shader (“BlBasicEffectClipColor.mgfxo” and “BlBasicEffectClipColorOGL.mgfxo” for OpenGL), which “creates” its own alpha channel from a specified texture color. Use it with non-translucent textures for which you want some translucency. Use it like BlBasicEffectAlphaTest but instead of setting the AlphaTestThreshold variable, set the ClipColor and ClipColorTolerance variables. ClipColor is the texture color that should indicate transparency (a Vector3 or Vector4), and ClipColorTolerance is a float that indicates how close to ClipColor (0 to .999) the texture color must be to cause transparency. BlBasicEffectClipColor is especially useful for videos that neglected to include an alpha channel.

See the [Custom effects](#_Custom_effects) section for details on using a custom effect.

# Texture transforms

The BlBasicEffectAlphaTestXformTex shader (“BlBasicEffectAlphaTestXformTex.mgfxo” and “BlBasicEffectAlphaTestXformTex OGL.mgfxo” for OpenGL) does the same thing as BlBasicEffectAlphaTest, but adds a feature that let’s you transform the texture.

Parameters are AlphaTestThreshold (same as used by the BlBasicEffectAlphaTest shader), TextureTranslate (a Vector2 that translates the texture), and TextureTransform (a 2x2 matrix that transforms the texture, specified as a Vector4 because there is no 2x2 matrix in MonoGame).

See the TextureTransform example and the [Custom effects](#_Custom_effects) section for details.

(Note: To make room for the extra arithmetic operations required by this shader, the code from the original BasicEffect for pixel lighting [an advanced form of bump mapping] has not been included in this shader.)

# Setting and dynamically changing a sprite’s scale, orientation, and position

Each sprite has a “Matrix” member that defines its orientation, scale, position, etc. relative to its parent sprite, or to an unmodified coordinate system if there is no parent. There are many static and instance methods of the Matrix class that let you easily set and change the scaling, position, rotation, etc. of a matrix.

When you change anything about a sprite’s matrix, you also change it for the child sprites, if any. That is, subsprites reside in the parent sprite’s coordinate system. For example, if a child sprite’s matrix scales it by 3, and its parent sprite’s matrix scales by 4, then the child sprite will be scaled by 12 in world space. Likewise, rotation, shear, and position are inherited, as well.

There are also static and instance Matrix methods and operator overloads to “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 because matrix multiplication is not commutative. 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 following [Matrix internals](#_Matrix_internals) 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 initial value, and X’ is the final value)

… which, when applied to each vertex, 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 constants (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 about the origin), then use the matrix:

3 0

0 1

The following matrix flips (mirrors) the model vertically about the origin:

1 0

0 -1

Below is a matrix to rotate a model counterclockwise by 90 degrees about the origin:

0 -1

1 0

Here is a matrix that rotates a model counterclockwise by 45 degrees about the origin:

0.707 -0.707

0.707 0. 707

Note that ‘0.707’ is the sine of 45 degrees, or cosine of 45 degrees.

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.)

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

Rather than using the above 16 letters (‘a’ through ‘p’) for the matrix elements, 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! The Matrix class provides matrix multiply, inversion, etc. methods. If you are interested in how the individual matrix elements are processed to perform matrix arithmetic, please look it up online.

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 with the child’s matrix to create the final (“absolute”) matrix used to draw that child, and that matrix is also used as the parent matrix for the subsprites of that child.

# A Short Glossary of 3D Graphics Terms

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".

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. Pixels across the face of a polygon are (typically) interpolated from the vertex color, texture, and normal values.

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.

Origin

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

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 numbers that can describe a difference, or transform, in one coordinate system from another. Each sprite has a matrix that defines its location, rotation, scale, shear etc. within the coordinate system of its parent sprite, or within an untransformed coordinate system if there is no parent. See [Dynamically changing a sprite’s orientation and position](#_Dynamically_changing_a).

Frame

In this document, 'frame' is analogous to a movie frame. A moving 3D scene is created by drawing successive frames.

Depth buffer

3D systems typically keep track of the depth of the polygon surface (if any) at each 2D window pixel so that they know to draw the nearer pixel over the farther pixel in the 2D display. The depth buffer is an array with one element per 2D window pixel, where each element is (typically) a 32-bit floating point value indicating the nearest (to the camera) depth of that point. In that way pixels that are farther away need not be drawn. You can override this behavior for special cases. See BlGraphicsDeviceManager.NearClip, BlGraphicsDeviceManager.FarClip. and search the web for MonoGame depth information.

Near clipping plane (NearClip)

The distance from the camera at which a depth buffer element is equal to zero. Nearer surfaces are not drawn.

Far clipping plane (FarClip)

The distance from the camera at which a depth buffer element is equal to the maximum possible floating-point value. Farther surfaces are not drawn.

Model space

The untransformed three-dimensional space that models are initially created/defined in. Typically, a model is centered on the origin of model space.

World space

The three-dimensional space that you see through the two-dimensional view of the window. A model is transformed from model space to world space by its final matrix (that is, the matrix we get *after* a sprite’s matrix is multiplied by its parent sprite matrices, if any).

View space

The two-dimensional space of the window on the screen. Objects in world space are transformed by the view matrix and projection matrix to produce the contents of the window. You don’t have to understand the view and projection matrices, though, because there are higher-level functions that control them—like Zoom, aspect ratio, and camera position and orientation functions.

# Troubleshooting

Q: When I set a billboard attribute of a flat sprite (like a plane), I can no longer see it.

A: Perhaps the billboard orientation is such that you are looking at the plane from the side or back. Try setting a rotation in the sprite’s matrix (and make sure it doesn’t just rotate it on the axis intersecting your eye point).

Q: When I’m inside a sprite, I can’t see it.

A: By default, Blotch3D draws only the outside of a sprite. Try putting a "Graphics.GraphicsDevice.RasterizerState = RasterizerState.CullClockwise” (or set it to CullNone to see both the inside and outside) in the BlSprite.PreDraw delegate, and set it back to CullCounterClockwise in the BlSprite.DrawCleanup delegate.

Q: I set a sprite’s matrix so that one of the dimensions has a scale of zero, but then the sprite, or parts of it, become black.

A: A sprite’s matrix also affects its normals. By setting a dimension’s scale to zero, you may have caused some of the normals to be zeroed-out as well. Try setting the scale to a very small number, rather than zero.

Q: When I am zoomed-in a large amount, sprite and camera movement jumps as the sprite or camera move.

A: You are experiencing floating point precision errors in the positioning algorithms. About all you can do is “fake” being that zoomed in by, instead, moving the camera forward temporarily. Or simply don’t allow zoom to go to that extreme.

Q: Sometimes I see slightly farther polygons and parts of polygons of sprites appear in front of nearer ones, and it varies as the camera or sprite moves.

A: The floating-point precision limitation of the depth buffer can cause this. Disable auto-clipping on one or both of NearClip and FarClip, and otherwise try increasing your near clip and/or decreasing your far clip so the depth buffer doesn’t have to cover so much dynamic range.

Q: I have a sprite that I want always to be visible, but I think its invisible because its outside the depth buffer, but I don’t want to change the clipping planes (NearClip and FarClip).

A: Try doing a "Graphics.GraphicsDevice.DepthStencilState = Graphics.DepthStencilStateDisabled” in the BlSprite.PreDraw delegate, and set it back to DepthStencilStateEnabled in the BlSprite.DrawCleanup delegate.

Q: I’m moving or rotating a sprite regularly over many frames by multiplying its matrix with a matrix that represents the change per frame, but after a while the sprite gets distorted or drifts from its predicted position, location, rotation, etc.

A: When you multiply two matrices, you introduce a very slight floating-point inaccuracy in the resulting matrix because floating-point values have a limited number of bits. Normally the inaccuracy is too small to matter. But if you repeatedly do it to the same matrix, it will eventually become noticeable. Try changing your math so that a new matrix is created from scratch each frame, or at least created every several hundred frames. For example, let’s say you want to slightly rotate a sprite every frame by the same amount. You can either create a new rotation matrix from scratch every frame from a simple float scalar angle value you are regularly incrementing, or you can multiply the existing matrix by a persistent rotation matrix you created initially. The former method is more precise, but the latter is less CPU intensive because creating a rotation matrix from a floating-point angle value requires that transcendental functions be called, but multiplying matrices does not. A good compromise is to use a combination of both, if possible. Specifically, multiply by a rotation matrix most of the time, but on occasion recreate the sprite’s matrix directly from the scalar angle value.

# Rights

Blotch3D (formerly GWin3D) Copyright (c) 1999-2018 Kelly Loum, all rights reserved except those granted in the following license:

Microsoft Public License (MS-PL)

This license governs use of the accompanying software. If you use the software, you

accept this license. If you do not accept the license, do not use the software.

1. Definitions

The terms "reproduce," "reproduction," "derivative works," and "distribution" have the

same meaning here as under U.S. copyright law.

A "contribution" is the original software, or any additions or changes to the software.

A "contributor" is any person that distributes its contribution under this license.

"Licensed patents" are a contributor's patent claims that read directly on its contribution.

2. Grant of Rights

(A) Copyright Grant- Subject to the terms of this license, including the license conditions and limitations in section 3, each contributor grants you a non-exclusive, worldwide, royalty-free copyright license to reproduce its contribution, prepare derivative works of its contribution, and distribute its contribution or any derivative works that you create.

(B) Patent Grant- Subject to the terms of this license, including the license conditions and limitations in section 3, each contributor grants you a non-exclusive, worldwide, royalty-free license under its licensed patents to make, have made, use, sell, offer for sale, import, and/or otherwise dispose of its contribution in the software or derivative works of the contribution in the software.

3. Conditions and Limitations

(A) No Trademark License- This license does not grant you rights to use any contributors' name, logo, or trademarks.

(B) If you bring a patent claim against any contributor over patents that you claim are infringed by the software, your patent license from such contributor to the software ends automatically.

(C) If you distribute any portion of the software, you must retain all copyright, patent, trademark, and attribution notices that are present in the software.

(D) If you distribute any portion of the software in source code form, you may do so only under this license by including a complete copy of this license with your distribution. If you distribute any portion of the software in compiled or object code form, you may only do so under a license that complies with this license.

(E) The software is licensed "as-is." You bear the risk of using it. The contributors give no express warranties, guarantees or conditions. You may have additional consumer rights under your local laws which this license cannot change. To the extent permitted under your local laws, the contributors exclude the implied warranties of merchantability, fitness for a particular purpose and non-infringement.