# Blotch3D

# Features

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

You can...

* Convert and then load standard 3D model file types as “sprites” and display and move thousands of them in 3D at high frame rates.
* Programmatically create a wide variety of sprite shapes.
* Create sprites by defining individual polygons.
* Load textures from standard image files, including textures with an alpha channel (i.e. with translucent pixels).
* Set a sprite’s material, texture, and lighting response.
* Show 2D and in-world 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 real-time 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 on a sprite (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’.
* Implement GUI controls in the 3D window as dynamic 2D text or image rectangles, and with transparent pixels.
* 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 a texture on a surface.
* Use with WPF Core and WinForms Core. (Also see <https://github.com/sqrMin1/MonoGame.Forms>)
* Detect sprite radius collisions.
* Implement fog.
* Create totally configurable particle systems.
* Define ambient lighting and up to three point-light sources.
* Several shaders are provided to support texture transforms, alpha textures with lighting, etc.
* Easily write your own custom shaders using the provided shader code as a template.
* All other MonoGame features remain available.
* Library is .NET Standard
* You can develop Blotch3D apps on Windows, MacOS, and Linux. Blotch3D apps can be built for any Microsoft Windows platforms, iOS, Android, MacOS, Linux, PS4, PSVita, Xbox One, and Switch.

# Overview

Blotch3D sits on top of MonoGame and all MonoGame’s features are still available. MonoGame is a widely used 3D library for C#. It is open source, 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 “Blotch3D.chm”. Note: To support Doxygen documentation generator, 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. 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 there are certain limitations in MonoGame. Currently you can only have one 3D window per process. (Creating multiple 3D windows can be buggy—unless you do it from separate processes.) 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.

Several bare-bones examples demonstrate some of the more common tasks with just a few lines of code. The source file name for each example project is Example.cs. You can do a diff between the basic example’s Example.cs and another example’s Example.cs to see what extra code must be added to implement the features it demonstrates.

# Quick start for Windows

To deliver a Blotch3D app for Windows, just deliver the contents of your project’s output folder. No other software need be installed on the target system.

To develop with Blotch3D…

1. Download the Blotch3D repository, or clone it.
2. Open the Visual Studio solution file (Blotch3D.sln) and build and run the example projects.
3. Use IntelliSense or see “Blotch3D.chm” for the reference documentation.
4. See [Creating a new project](#_Creating_a_new) for details on creating projects, adding Blotch3D to an existing project, or building for another platform.

To convert existing models, create font descriptions, etc., you’ll need to use the Monogame Pipeline tool. See <https://docs.monogame.net/articles/tools/mgcb.html>. The docs well describe how to use it, but in essence you do this:

1. Run it from the command line with: mgcb\_editor
2. Use the ‘New’ menu item to create a new project (mgcb) file
3. Open a model by right-clicking the project in the ‘Project’ pane and selecting the ‘Add’ item
4. Build the items by right-clicking the project in the ‘Project’ pane and selecting rebuild. Note the output folders in the ‘Properties’ pane.

To create new shaders you’ll need the Monogame effects compiler. See <https://docs.monogame.net/articles/tools/mgfxc.html>. See the make\_effects.bat file for examples of building the existing shaders.

If the content manager (pipeline tool) complains of not having a DLL, get the installer for “Visual C++ Redistributable for Visual Studio 2012” for your platform from <https://www.microsoft.com/en-us/download/details.aspx?id=30679> and run it with the default settings. Later versions (than 2012) do not have the necessary DLL.

# Deficiencies and Alternatives

Although any feature can certainly be implemented by the app developer, notable features directly lacking in Blotch3D/MonoGame are…

* Shadows (although they might be added in the future)
* Physics
* Per-face collision detection
* Optimized (tree) collision detection
* More than one 3D window per process
* A NuGet package

I haven’t investigated it, but MonoGame.Extended seems to provide some but not all of the same features as Blotch3D. You could even try using both together, but if you do that then make sure they were built with the same version of Monogame, and I don’t know any details on how you would share things like the MonoGame ‘Game’ object and the GraphicsDeviceManager object. See [https://www.monogameextended.net](https://www.monogameextended.net/).

Another alternative is UrhoSharp. I haven’t looked at it in detail, but below are listed some of its advantages.

* UrhoSharp has a NuGet package
* UrhoSharp supports physics
* UrhoSharp supports octree collision detection
* UrhoSharp supports shadows

A few UrhoSharp disadvantages (compared to Blotch3D) I happened to notice are:

* UrhoSharp bare bones code is a bit more complicated than Blotch3D’s
* The official UrhoSharp reference documentation is sparse or non-existent
* Although there are third party help and discussions for UrhoSharp, there is notably more for MonoGame (Blotch3D’s underlying 3D engine)
* UrhoSharp is notably younger than MonoGame. There seemed to be more recent bug reports.
* UrhoSharp supports less or no programmatic object creation
* There doesn’t appear to be an intrinsic texture transform shader
* Particle systems are not as versatile

Finally, a mention of three.js is in order, even though it is JavaScript rather than C#, because it’s so full featured yet easy to learn. See <https://docs.microsoft.com/en-us/windows/uwp/get-started/get-started-tutorial-game-js3d> for Visual Studio-based development of three.js.

# Creating a new project

(See the [Quick start for Windows](#_Quick_start_for) section to set everything up for development)

You can create a new .NET Core/WPF Core/Windows Forms Core project and add a reference to Blotch3D.

To add MonoGame + Blotch3D to an existing Windows Core project, add a reference to Blotch3D.

For .NET Framework projects, add a reference to Blotch3D and also add the NuGet package ‘MonoGame.Framework.DesktopGL’

To create a project for another platform besides Microsoft Windows: First 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. Then use the MonoGame Visual Studio project wizard to create a project for that platform that will be the Blotch3D class library. Delete any default source files created by the wizard and add the source files of the Blotch3D library. Go to project properties and change the project type from an executable to a class library. Then use the same wizard to create a project for that same platform that will be your app and add to it a reference to that Blotch3D project you created first. For some platforms you may need to do some online research to properly create projects.

# Development overview

See the examples, starting with the basic example.

You define a 3D window by deriving a class from BlWindow3D and overriding at least the FrameDraw method. Open the window by instantiating that class and calling its “Run” method *from the same thread*. The Run method then calls the methods you’ve overridden, when appropriate, and does not return until the window has closed.

All code that accesses the 3D hardware must be in BlWindow3D overridden methods. This is because 3D subsystems (OpenGL, DirectX, etc.) generally require that a single thread access all 3D hardware resources for a given 3D window. There are certain platform-specific exceptions to this rule, but we don’t use them. This rule also applies to any code structure (like Parallel, etc.) that may internally use other threads, as well. Also, since sometimes it’s hard to know exactly what 3D operations really do hit the 3D hardware, it’s best to assume all of them do, like creation and use of all Blotch3D and MonoGame objects.

You can put all your 3D code in the one overridden method called “FrameDraw”, if you like, but there are a couple of other overridable methods provided for your convenience. There is a Setup method that is called once when the 3D window is first opened and a FrameProc method that is called every frame. The FrameDraw method is also called each frame, but only when there is enough CPU available. You are welcome to put whatever you like in any of those three methods, except that actual drawing code (code that causes things to appear in the window) must be in the FrameDraw method.

For apps that may suffer from severe CPU exhaustion (at least for the 3D thread), it might be best to put all your periodic 3D code in FrameDraw and not bother with FrameProc. In this way your code will be called less often under high-CPU loads. Of course, then your periodic code should handle being called at a variable rate.

You can also specify a delegate to the BlSprite constructor. The delegate will also be executed every frame. The effect is the same as putting the code in FrameProc, but it better encapsulates sprite-specific code.

A single-threaded application would have all its code in the overridden methods or delegates. If you are developing a multithreaded program, then you would probably want to reserve the 3D thread (the call to the BlWindow3D’s ‘Run’ method and its overrides) 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 to the 3D thread with BlWindow3D.EnqueueCommand or BlWindow3D.EnqueueCommandBlocking, which will be executed within one frame time by the 3D thread.

Models or VertexBuffers must be added to the BlSprite.LODs container for them to appear when you draw that sprite. Otherwise nothing is drawn. See how the examples create a model or vertex buffer and add it to that collection. When a sprite is disposed, it does not dispose the models in its LODs container. This is so you can add the same model to multiple sprites.

You can use a variety of methods to draw things in FrameDraw. 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 other sprites as its subsprites and call the Draw method of the Top sprite to cause the other sprites to be drawn. There are also 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). You can also draw things using the lower-level MonoGame methods. For example, it is faster to draw multiple 2D textures and text using MonoGame’s SpriteBatch class. See the ‘Making and Using 3D Models’ section for more information.

The easiest way to set the camera position and orientation is to periodically call Graphics.DoDefaultGui(). Typically, this is done in the FrameProc method, but could be done in the FrameDraw method as well. In any case call it before anything is drawn so that the camera is correct for subsequent drawing. If you want other ways to control the camera, then see the various Graphics.AdjustCamera… methods, the Graphics.SetCameraToSprite method, and the View, Eye, and LookAt fields.

BlWindow3D derives from MonoGame’s “Game” class, so you can also override other Game class overridable methods. Just be sure to call the base method from within a Game class overridden method.

Because multiple windows are not conducive to some of the supported platforms, MonoGame, and thus Blotch3D, do not support more than one 3D window in the same process. If you need multiple 3D windows, you’ll have to do it from multiple processes. You can *create* multiple 3D windows in the same process, but MonoGame does not handle them correctly (certain window input is sometimes ignored or goes to the wrong window, and in certain situations will cause an exception). This can be tested with the WPF example. You can, of course, create any number of non-3D windows you like in the same process.

Officially, Blotch3D+MonoGame must create the system window used for 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 BlWindow3D window be a child window of an existing GUI, you need to either explicitly size, position, and convey Z order to that 3D window so that it is overlaid over the child window, or use <https://github.com/sqrMin1/MonoGame.Forms>.

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

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

* The Models and VertexBuffers that you can add to BlSprite.LODs are MonoGame objects.
* The BlWindow3D class derives from the MonoGame “Game” class. The Setup, FrameProc, and FrameDraw methods are called by certain overridden Game methods. (Override MonoGame methods as you like but be sure to call the base method from within the overridden method.)
* 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, joysticks, etc.

Most Blotch3D and MonoGame objects must be Disposed when you are done with them and you are not otherwise terminating the program. And they must be disposed by the same thread that created them. You’ll get an informative exception by the finalizer if this isn’t done.

See the examples, reference documentation (doc/Blotch3D.chm), and IntelliSense for more information.

# Making and using 3D models

You can use the BlGeometry static class to make a variety of objects programmatically. See the geometry examples and that class for more information. A few primitive models are also included with Blotch3D. They can be used as is done in the examples that use them if the Blotch3D project is included in your solution and the reference’s ‘Copy local’ flag is set, or just add the XNB file to your project and set its ‘copy if newer’ or ‘copy always’ option.

You can also convert standard 3D model files, fonts, etc. to “XNB” files for use by your MonoGame project. The MonoGame content manager (pipeline tool) is used to make this conversion. See the [Quick start for Windows](#_Quick_star)  section for details.

There are countless standard 3D models that can be downloaded, or you can create one from scratch. To create one from scratch, if you don’t need a texture image on the model you may be able to get by with the ultra-simple Tinkercad on-line modeler. It takes only a few minutes to learn to use it. If you do need an image texture on the model or other more advanced features, you can use Blender. Blender is a full featured and free professional modeler, but with a fairly steep learning curve. To create the texture map with Blender, see 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> .

Since typically standard file types need to be converted to XNB files only once, one can consider it a separate manual step that should be done immediately after creating, choosing, or changing the standard resource during development. For example, after downloading or creating a 3D model, run it through the pipeline manager to create your XNB file. Then add that XNB file to your project and set its project properties so it is copied to the output folder for loading at run time.

You can even programmatically call the pipeline tool, or even call methods directly in the MonoGame.Framework.Content.Pipeline.dll to do the conversion at run-time, but only on target platforms that would also support development. See <https://community.monogame.net/t/building-and-loading-content-at-runtime/10849> for more information. (And speaking of importing files programmatically, you can use the BlGraphicsDeviceManager.LoadFromImageFile method to load image files directly, probably on most any platform.)

# Particles

Particle systems in Blotch3D are implemented by specifying BlSprite.FrameProc delegates. So, particles systems are completely configurable. For example, you can implement nonlinear or abrupt changes in the particle’s life or make particle trees (particles with particles). See the Particle example.

# Custom effects

By default, Blotch3D draws sprites using a standard shader that comes with MonoGame which is managed by a MonoGame BasicEffect object.

Blotch3D also provides several custom shaders that are the same as that managed by BasicEffect, but they provide added features. To use them, you instantiate a BlBasicEffect, pass the shader file name to its constructor, and set it with the SetEffect delegate of BlSprite. Example source code is shown below, and working examples are provided that demonstrate how to use several such custom shaders.

The custom compiled shader files for DirectX and OpenGL are in the src/Content/Effects folder. See below for compiling for different platforms. To use a custom shader, first copy the compiled shader file (mgfxo file) to your program’s output folder—you might add a link to it in your project and set its build properties so it is copied to the output folder when your project builds.

When your program runs, it specifies that file name in the BlBasicEffect constructor (or you can manage the bytes from the file, yourself, and pass the bytes to the constructor). Then when the sprite is drawn, the effect must be specified by the sprite’s SetEffect delegate.

A BlBasicEffect supports the several material and lighting parameters that are gotten from the BlSprite material and lighting fields with a call to BlSprite.SetupBasicEffect. But besides those, each effect also typically has certain other parameters that must be specified that control the unique feature(s) provided by the custom shader. These are set with the BlBasicEffect.Parameters[].SetValue method. They can be set at any time.

For example, the BlBasicEffectAlphaTest shader is used like this:

// Create a 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 above which pixels should be drawn.

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

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

// Specify a SetEffect delegate that sets 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 in the same folder as the compiled shader files. It’s just a copy of the original MonoGame BasicEffect shader code, but with a few lines added. To compile the shaders, be sure to add the path to 2MGFX.exe to the ‘path’ environment variable. Typically, the path is “\Program Files (x86)\MSBuild\MonoGame\v3.0\Tools”. Then run the make\_effects.bat file.

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.

Documentation for individual custom shaders follows.

# Translucency with the BlBasicEffectAlphaTest shader

Each pixel of a texture has a red, a green, and 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 chronologically drawn *before* anything farther away, because drawing a surface also updates the depth buffer (see Depth Buffer in the glossary). Since the depth buffer records the nearer pixel, it prevents farther 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 the artifacts aren’t visible. (Note: subsprites are drawn in the order of their names.)

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 translucent sprites 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 and still may exhibit various artifacts. 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 managed with 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 with the BlBasicEffectClipColor shader

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 (specifically, it’s a threshold of the square of the difference between pixel color and ClipColor). BlBasicEffectClipColor is especially useful for videos that neglected to include an alpha channel.

See the [Translucency with the BlBasicEffectAlphaTest shader](#_Translucency_with_the) section for an introduction to alpha and alpha testing, and see the [Custom effects](#_Custom_effects) section for details on using a custom effect.

# Transforming textures with the BlBasicEffectAlphaTestXformTex shader

The BlBasicEffectAlphaTestXformTex shader (“BlBasicEffectAlphaTestXformTex.mgfxo” and “BlBasicEffectAlphaTestXformTexOGL.mgfxo” for OpenGL) does the same thing as BlBasicEffectAlphaTest but adds a feature that lets you transform the texture on the surface of the sprite.

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 required extra arithmetic operations, the code from the original BasicEffect for pixel lighting [an advanced form of bump mapping] has been removed from this shader.)

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

Each sprite has a “Matrix” member. The Matrix member defines the sprite’s orientation, scale, position, etc. relative to its parent sprite, or to an unmodified coordinate system if there is no parent.

When you change anything about a sprite’s matrix, all its descendants (its subsprites, and their subsprites, etc.) automatically follow that change. 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. In this way you can construct complex objects (sprite trees) and not worry about each individual part as you alter the orientation and position of that object (i.e. the topmost sprite).

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. For example, to set the scale, assign the sprite’s Matrix to Matrix.CreateScale(). You can also create matrices by multiplying existing matrices or create them piecemeal.

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 as desired, it can be done 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

Read this section only if you want a deeper understanding of altering a sprite’s orientation, position, etc. than is explained in the previous section.

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 (0.2, 0.1) to each of those original vertices, which would result in final model vertices of (4.2, 1.1) and (3.2, 3.1). 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 also 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. (And since matrixes can be combined by multiplying them, we can, for example, rotate a model before it has been translated, and then translate it so that it rotates locally around its own model 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 of the vertices.

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. Specifically, it depends on the normals of the vertices of that polygon, and 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 value that varies between (0, 0) and (1, 1) which indicates a position within the 2D bitmap image associated with that sprite, that should be aligned with that vertex. Pixels across the surface of a polygon are interpolated from the texture coordinates specified for each vertex. A texture’s (X,Y) coordinate is referred to as its (U,V) coordinate to discriminate it from 3D coordinates of vertices.

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, because of the way the brain interprets an image. 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 in an untransformed coordinate system.

Y-axis

The axis that extends forward from the origin in an untransformed coordinate system.

Z-axis

The axis that extends up from the origin in an untransformed coordinate system.

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.

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 last drawn nearest (to the camera) depth of that point. In that way pixels that are farther away need not and should not be drawn. NearClip defines what a depth value of zero means (the nearest distance kept track of), and FarClip defines what a depth value of the maximum floating point value means (the farthest). Objects outside that range are not drawn. If the range is too large, then limited floating point resolution in the 32-bit distance value will cause artifacts. See the troubleshooting question about depth. You can disable the depth testing for special cases (see the troubleshooting question about disabling the depth buffer). See BlGraphicsDeviceManager.NearClip, BlGraphicsDeviceManager.FarClip. and search the web for MonoGame depth information.

Near clipping plane (BlGraphicsDeviceManager.NearClip)

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

Far clipping plane (BlGraphicsDeviceManager.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. See ‘Depth buffer’.

Model space

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

World space

The three-dimensional space that you see through the two-dimensional screen 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 or made invalid. Try setting the scale to a very small number, rather than zero.

Q: When I am zoomed-in a very 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 or set limits on auto-clipping in 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. In some cases you might disable the depth buffer. See the answer to the question on a sprite being outside the depth buffer.

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

A: Try disabling the depth buffer just for that sprite with 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 (to reduce CPU usage). 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, or you can multiply the existing matrix by a persistent rotation matrix you created initially. The former method is more precise but takes a lot more CPU, and latter is less CPU intensive but suffers from the eventual visible floating point innaccuracy. A good compromise is to use a combination of both, if possible. Specifically, multiply by the same rotation matrix most of the time, but on occasion recreate the sprite’s matrix directly from the scalar angle value that you’ve been incrementing.

Q: I’m using SetCameraToSprite to implement cockpit view, but when the sprite moves, the camera lags from the sprite’s position.

A: It’s a chicken and egg problem. The sprite must be moved *before* moving the camera to its position, but the camera must be moved *before* showing the sprite’s latest position. The only way to fix this is to set the sprite’s position without showing (drawing) it, then call SetCameraToSprite, then draw everything. If you want to attach the camera to a child sprite, you might want to disable any time-consuming tasks in drawing things when you only want to calculate the sprite’s position without drawing it, then when it comes time to draw things, enable them.

# Rights

Blotch3D (formerly GWin3D) Copyright (c) 1999-2020 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.