# Blotch3D

Create real-time 3D graphics with just a few lines of C# code.

# Quick start

1. Get the installer for the latest release of MonoGame from <http://www.monogame.net/downloads/> and run it. (Do NOT get the current development version nor the NuGet package.)
2. Get the Blotch3D repository zip from <https://github.com/Blotch3D/Blotch3D> and unzip it.
3. Open the Visual Studio solution file (Blotch3D.sln).
4. Build and run the example projects. (For other platforms, you’ll need the appropriate Visual Studio add-on and you will need to create a separate project for that platform.)
5. Use IntelliSense to see the reference documentation, or see “Blotch3DManual.pdf”.

# Introduction

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

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 (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.
* You can give the user easy control over all aspects of the camera (zoom, pan, truck, dolly, rotate, etc.).
* Easily control all aspects of the camera programmatically.
* Create billboard sprites.
* 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, to implement weapons fire, etc.
* Get a list of sprites under the mouse position, to implement mouse selection, tooltips, pop-up menus, etc.
* Implement levels-of-detail.
* Implement mipmaps.
* 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 collisions between sprites.
* Implement fog
* Define ambient lighting, and up to three point-light sources. (More lights can be defined if a custom shader is used.)
* 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.

# Project structure

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 the Microsoft Windows x64 platform. See below for other platforms.

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

To create a new project, you must first install MonoGame as described in the [Quick start](#_Quick_start) section, if you haven’t already. You must also install the Visual Studio add-ons, etc. for the desired platform if different from Microsoft Windows. (For example, for Android you’d need the Xamarin for Android add-on.)

For Microsoft Windows, you can create a new project by either copying an existing Blotch3D example project and renaming it, or you can use the project wizard to create a MonoGame project and then add a reference to Blotch3D or the Blotch3D source.

For other platforms, you can look online for instructions on creating a MonoGame project/platform type you want and then add a reference to, or the source of, Blotch3D.

Or you can:

1. Create a new project with the project wizard that is close to the type you want or use online instructions for creating it.
2. Add a reference to MonoGame if it doesn’t already have one. (typically found in \Program Files (x86)\MonoGame\v3.0\Assemblies\...)
3. Include the Blotch3D source in the project, or a Blotch3D project in the solution, or add a reference to a build of it for that platform.
4. Follow the procedure in the ‘[Making 3D models](#_Making_3D_models)’ section to add a content folder and the pipeline manager so that you have a way to add content.
5. If available on the selected platform, while debugging you’ll probably want to temporarily set the output type to a type that shows stdout messages (like ‘Console Application’ on Microsoft Windows) so you can see any debug messages.
6. You may need to copy various XML structures into your csproj file from other projects that have some of the attributes that you want.

To create a 3D window, follow the guidelines in the [Development](#_Development) section.

# Development

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

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 BlWindow 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”.

All code that accesses 3D hardware resources must be done in the 3D thread, including code that creates and uses all Blotch3D and MonoGame objects. Note that this rule also applies to any code structure (Parallel, async, etc.) that may internally use other threads, as well. This is necessary because certain 3D subsystems (OpenGL, DirectX, etc.) generally require that 3D resources be accessed by a single thread. (There are some platform-specific exceptions, but MonoGame does not use them.)

This pattern and these rules are also used by MonoGame. In fact, the BlWindow3D class inherits from MonoGame’s “Game” class. But instead of overriding certain “Game” class methods, you override BlWindow3D’s Setup, FrameProc, and FrameDraw methods. Other “Game” class methods and events can still be overridden, if needed.

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 of instantiation of the BlWindow3D-derived object. 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. (Note: You can also pass a delegate to the BlSprite constructor, which will cause that delegate to be executed every time the BlWindow3D’s FrameProc method is executed. The effect is the same as putting the code in FrameProc, but it better encapsulates sprite-specific code.)

The FrameDraw method is called by the 3D thread every frame, but only if there is enough CPU for that thread. Otherwise it is called less frequently. This is where you must 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 want to put your app code in this method so it is called less frequently (as long as that code can properly handle being called at variable rates).

A single-threaded application would have all its code in those three overridden methods.

If you are developing a multithreaded app, then you would probably want to reserve the 3D thread only for tasks that access 3D resources. When other threads do need to create, change, or destroy 3D resources or otherwise do something in a thread-safe way with the 3D thread, they can pass a delegate to 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 window. You *can* create multiple windows, but they don’t work 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. (On Microsoft Windows, you can use the BlWindow3D.Form object for that.)

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 original MonoGame window so that it is overlaid over the child window.

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. So, you can, for example, specify custom shaders, etc., for those models.
* The BlWindow3D class derives from the MonoGame “Game” class.
* The BlGraphicsDeviceManager class derives from MonoGame’s “GraphicsDeviceManager” class.

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, and IntelliSense for more information.

# Translucency

Translucent pixels in text or textures drawn using the 2D Blotch3D drawing methods (BlGraphicsDeviceManager#DrawText and BlGraphicsDeviceManager#DrawTexture) will always correctly show the things behind them. Just be sure to call those methods after all other 3D things are drawn.

However, a translucent texture applied to a sprite may require special handling.

If you simply apply the translucent texture to a sprite as if it’s just like any other texture, there may be situations that you will see certain undesirable artifacts depending on whether a far surface with respect to the camera is drawn before or after a near surface. For some translucent textures the artifacts can be subtle, and you might find them perfectly acceptable. We do this in the “full” example because the draw order is 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.

One main reason these artifacts occur is because the default MonoGame “Effect” used to draw models (the “BasicEffect” effect) provides a pixel shader that does not do “alpha testing”. Alpha testing is the process of neglecting to draw texture pixels if the texture pixel’s alpha is below some value (i.e. if it is translucent enough). (Although even then, there are artifacts that occur when multiple *partly*-translucent textures occlude one another, but that is a subject for a more advanced text.)

MonoGame does provide a separate “AlphaTestEffect” effect that supports it. But AlphaTestEffect does not support directional lights, as are supported in BasicEffect. So, don’t bother with AlphaTestEffect.

For these reasons Blotch3D includes a custom effect called BlBasicEffect that provides everything that MonoGame’s BasicEffect provides, but also provides alpha testing. See the SpriteAlphaTexture example. Specifically, you do the following:

1. Copy the “BlBasicEffect.mgfxo” file from the Blotch3D source to your program execution folder.
2. Your program loads that file and creates a BlBasicEffect, like this:

byte[] bytes = File.ReadAllBytes("BlBasicEffect.mgfxo");

BlBasicEffect = new BlBasicEffect(Graphics.GraphicsDevice, bytes);

1. And it specifies the alpha level that merits drawing the pixel, like this (this could also be done in the delegate described below):

BlBasicEffect.Parameters["AlphaTestThreshold"].SetValue(.2f);

1. And then your program assigns a delegate to the BlSprite’s SetEffect field (for sprites that have translucent textures). The delegate does something like this:

MySprite.SetEffect = (s,effect) =>

{

s.SetupBasicEffect(BlBasicEffect);

return BlBasicEffect;

};

Note that BlBasicEffect is slightly slower than the default effect, so only use BlBasicEffect when necessary.

# Making 3D models

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

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

If the “MonoGameContentReference” build option is not available in the drop-down list because, for example, you have created a project from scratch (rather than copied an existing example), then try this:

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

1. Open your application .csproj in an 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" />

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

If no existing model meets your needs, you can either programmatically create a model by specifying the vertices and normals (see the example that uses custom Vertices), or create a model with, for example, the Blender 3D modeler and then add it to the project with the pipeline manager. The pipeline manager can import several model file types. 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> . Also, you may be able to import certain existing models from the web, but mind the copyright.

# Dynamically changing a sprite’s orientation and position

Each sprite has a “Matrix” member that defines its orientation and position relative to its parent sprite, 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, translation, rotation, etc. of a matrix.

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

There are also static and instance Matrix methods and operator overloads to combine (multiply) matrices to form a single matrix which combines the effects of multiple matrices. For example, a rotate matrix and a scale matrix can be multiplied to form a single rotate-scale matrix. But mind the multiplication order 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 rest of this section should be studied only when you need a deeper knowledge.

# Matrix internals

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

Let’s imagine a model that has one vertex at (4,1) and another vertex at (3,3). (This is a very simple model comprised of only two vertices!)

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

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

If we want to scale (stretch) the X relative to the origin, we can multiply the X of each vertex by 2.

For example,

X’ = 2X (where X is the 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 elements (a, b, c, and d) in a 2x2 matrix, like this:

a b

c d

This should all be very easy to understand.

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

For example, if we apply the following matrix to each of the model’s vertices:

1 0

0 1

…then the vertices are unchanged, because…

X’ = 1X + 0Y

Y’ = 0X + 1Y

…sets X’ to X and Y’ to Y.

This matrix is called the *identity* matrix because the output (X’,Y’) is the same as the input (X,Y).

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

3 0

0 3

To scale only X by 3 (stretch a model in the X direction 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.

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

Vertex

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

Polygon

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

Ambient lighting

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

Diffuse lighting

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

Texture

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

Normal

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

X-axis

The axis that extends right from the origin.

Y-axis

The axis that extends forward from the origin.

Z-axis

The axis that extends up from the origin.

Origin

The center of a coordinate system. The point in the coordinate system that is, by definition, at (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 depth of the last drawn pixel. In that way pixels that are farther away need not be drawn. You can override this behavior for special cases. See BlGraphicsDeviceManager.NearClip and BlGraphicsDeviceManager.FarClip.

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 doing 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 becomes 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 zero’d-out as well.

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 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. 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 depth buffer.

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 updating, 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 for a time, but somewhat periodically recreate the sprite’s matrix directly from the scalar angle value.

# Rights

Blotch3D (formerly GWin3D) is Copyright © 1999-2018 by Kelly Loum

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.