Blotch3D Manual

Contents

Chapter 1

Blotch3D

1.1 Quick start for Windows

On your development machine ...

1. Get the installer for the latest release of the MonoGame SDK for Visual Studio 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. Download the Blotch3D repository, or clone it.
3. Open the Visual Studio solution file (Blotch3D.sln) and build and run the example projects.
4. Use IntelliSense and see "Blotch3DManual.pdf" for the reference documentation.
5. See Creating a new project for details on creating projects, adding Blotch3D to an existing project, or building for another platform.
6. To deliver an app, just deliver the contents of your project's output folder.
7. Also see the Deficiencies section.

1.2 Features

Blotch3D is a C# library that vastly simplifies many of the tasks in developing real-time 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.
- 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 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, and see the description before using).
- 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.
- 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 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 "Blotch3DManual.pdf". 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/.

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 is 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.

"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 Example.cs files and another example's source file to see what extra code must be added to implement the features it demonstrates.

1.3 Deficiencies

Although any feature can certainly be implemented by the app developer, Blotch3D does not directly provide...

- 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

Also check out UrhoSharp before getting too heavily into developing with Blotch3D. I haven't looked at it in detail, but below are listed some differences between Blotch3D and UrhoSharp.

UrhoSharp advantages over Blotch3D that I've noticed:

- UrhoSharp has a NuGet package
- UrhoSharp supports physics
- UrhoSharp supports octree collision detection

- UrhoSharp supports shadows
- UrhoSharp supports Xamarin Forms (maybe Blotch3D does, also? —I just haven't tried it)

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

- 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

1.4 Creating a new project

To develop with Blotch3D, you must first install the MonoGame SDK as described in the `Quick start for Windows` section.

The easiest way to create a new project for Windows is to copy an existing example project (like the basic example) and then rename it using Visual Studio.

To add MonoGame + Blotch3D to an existing Windows 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: 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.

To distribute a program for Microsoft Windows, deliver everything in your project's output folder. Other platforms may require different delivery methods.

1.5 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, its 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 at the beginning 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 Frame↩ Draw 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 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.

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.

3D models must be added to the BlSprite.LODs container for them to appear when you draw that sprite. 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.

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. 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. On Microsoft Windows, you can also better control the window and add window event handlers with the associated Windows 'Forms' object, BlWindow3D.WindowForm.

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 (input sometimes goes to the wrong window and in certain situations will crash). 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 explicitly size, position, and convey Z order to that 3D window so that it is overlaid over the child window. The BlWindow3D.WindowForm field will be useful for this (Microsoft Windows only).

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, 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 if this isn't done.

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

1.6 Making and using 3D models

You can use the BlGeometry 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.

You can also convert standard 3D model files, fonts, etc. to "XNB" files for use by your MonoGame project. The MonoGame "pipeline manager" is used to make this conversion.

The Blotch3D project is already set up with the pipeline manager to convert the several primitive models to XNB files when Blotch3D is built. You can double-click "Content.mgcb" in the Blotch3D project to add more standard files and resources and to convert to XNB outside of the build process. You can also copy an XNB file to a project's output folder, where the program can load it.

When you create a new MonoGame project with the wizard, it sets up a "Content.mgcb" file in the new project that manages your content and runs the MonoGame pipeline manager as needed or when you double-click "Content.↩ mgcb" to add more content. That's fine for projects created with the project wizard. But it is a pain to add this feature to existing non-MonoGame projects, and certainly not necessary.

Since typically such 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 creating a 3D model with a 3D modeler, run it through the pipeline manager to create your XNB file, such as the one available from the Blotch3D project. 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. See http://www.↩ monogame.net/documentation/?page=MGCB for more information.

To create a new model file, it is recommended you use 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 .

1.7 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 tree structures. See the Particle example.

1.8 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 something like "\\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 follow.

1.9 Translucency with the BlBasicEffectAlphaTest shader

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, Bl↩ GraphicsDeviceManager#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 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 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 Alpha↩ TestEffect 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 Bl↩ BasicEffect 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 section and the SpriteAlphaTexture example for details.

1.10 Dynamically creating an alpha channel with the BlBasicEffectClipColor shader

Blotch3D includes a BlBasicEffectClipColor shader ("BlBasicEffectClipColor.mgfxo" and "BlBasicEffectClipColor↩ OGL.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 section for an introduction to alpha and alpha testing, and see the Custom effects section for details on using a custom effect.

1.11 Transforming textures with the BlBasicEffectAlphaTestXformTex shader

The BlBasicEffectAlphaTestXformTex shader ("BlBasicEffectAlphaTestXformTex.mgfxo" and "BlBasicEffectAlpha↩ TestXformTexOGL.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 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.)

1.12 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 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 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 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 section should be studied only when you need a deeper knowledge.

1.13 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 (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 matrix in the model coordinate system, and then translate it to a world coordinate system so that it rotates 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.

1.14 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. To discriminate a texture's (X,Y) coordinate from a vertex's 3D (X, Y, Z) coordinate, texture (X,Y) is more often called the texture's (U,V) coordinate.

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 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. See `Dynamically changing a sprite's orientation and position`.

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 be drawn. NearClip defines the nearest distance kept track of, and FarClip defines 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.

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.

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.

1.15 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.

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 just for that sprite (NearClip and FarClip).

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. 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 requires that transcendental functions be called, but multiplying existing 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.

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.

1.16 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.

Chapter 2

Namespace Index

2.1 Packages

Here are the packages with brief descriptions (if available):

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

Namespace Documentation

5.1 Blotch Namespace Reference

Classes

- class BlBasicEffect

  Holds a Blotch3D custom effect (like BlBasicEffectAlphaTest and BlBasicEffectClipColor) that is exactly like Mono↩ Game's BasicEffect but with certain added features. To make a BlBasicEffect, you must pass the mgfxo file name, or a byte array of its contents, to the constructor. See the SpriteAlphaTexture example and the section on Translucency for details on how to make and use objects of this class.
- class **BlDebug**

  This static class holds the debug flags. Many flags are initialized according to whether its a Debug build or Release build. Some flags enable exceptions for probable errors, and many flags cause warning messages to be sent to the console window, if there is one. For this reason you should first test your app as a debug build console app.
- class **BlEffectHelpers**

  Helper code shared between the various built-in effects.
- class BlGeometry

  Methods and helpers for creating various geometric objects. These methods create and manage regular (rectangular) grids of vertices as a flattened row-major array (if indexed as [y, x]) of VertexPositionNormalTexture[], triangle arrays (also as a VertexPositionNormalTexture[]), and VertexBuffers. You can concatenate multiple regular grids to produce one regular grid if they have the same number of columns, and you can concatentate multiple triangle arrays to produce one triangle array. If you are confused by this, see the examples. You can transform either type of array with TransformVertices. You can create smooth normals for a regular grid and facet normals for a triangle array. You can set texture (UV) coordinates. You can convert a regular grid to a triangle array. Finally, you can convert a triangle array to a VertexBuffer suitable for adding to a BlSprite.LODs field.
- class BlGraphicsDeviceManager

  This holds everything having to do with an output device. BlWindow3D creates one of these for itself. This derives from MonoGame GraphicsDeviceManager.
- class BlGuiControl

  A 2D GUI control. To create a GUI control: instantiate one of these, set its initial Texture, window position, and delegate, and then add it to BlWindow3D::GuiControls. (Any member can be dynamically changed.) The texture will be displayed, and then each frame the mouse is over it the delegate will be called. The delegate typically would examine the current mouse state (Mouse.GetState()) and the PrevMouseState member to detect button changes, etc. and perform an action. The delegate is called in the context of the window's 3D thread after the BlWindow3D::FrameProc method.
- class BlMipmap

  A BlMipmap holds a list of different resolutions of a texture, where one is applied to a sprite, depending on the ApparentSize of that sprite. You would assign it to a BlSprite::Mipmap. Note that this is a software mipmap (i.e. it isn't implemented in the 3D hardware). That is, unlike a hardware mipmap where different resolutions of the texture may appear on different parts of the sprite, only one resolution texture is used at time.

- class BlSprite

  A BlSprite is a single 3D object. Each sprite can also hold any number of subsprites, so you can make a sprite tree (a scene graph). Subsprites are drawn in the order of their sorted names. Child sprites 'follow' the orientation and position of the parent sprite. That is, they exist in the coordinate system of the parent sprite. The location and orientation of a sprite in its parent's coordinate system is defined by the sprite's Matrix member. Subsprites, LODs, and Mipmap are NOT disposed when the sprite is disposed, so you can assign the same one to multiple sprites.

- class BlWindow3D

  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 open 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, and not return until the window closes. All code that accesses 3D resources must be done in that thread, including code that creates and uses all Blotch3D and MonoGame objects. Note that this rule also applies to any code structure that may internally use other threads, as well. Do not use Parallel, async, etc. code structures that access 3D resources. Other threads that need to access 3D resources can do so by passing a delegate to EnqueueCommand and EnqueueCommandBlocking.

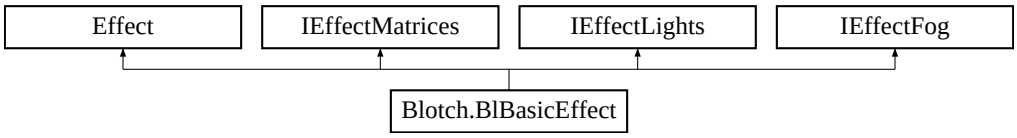
Chapter 6

Class Documentation

6.1 Blotch.BlBasicEffect Class Reference

Holds a Blotch3D custom effect (like BlBasicEffectAlphaTest and BlBasicEffectClipColor) that is exactly like Mono↩ Game's BasicEffect but with certain added features. To make a BlBasicEffect, you must pass the mgfxo file name, or a byte array of its contents, to the constructor. See the SpriteAlphaTexture example and the section on Translucency for details on how to make and use objects of this class.

Inheritance diagram for Blotch.BlBasicEffect:



Public Member Functions

- BlBasicEffect (GraphicsDevice device, string filename)

  Creates a new BlBasicEffect with default parameter settings. See class description for more info.
- BlBasicEffect (GraphicsDevice device, byte[ ] bytes)

  Creates a new BlBasicEffect with default parameter settings. See class description for more info.
- override Effect Clone ()

  Creates a clone of the current BasicEffect instance.
- void EnableDefaultLighting ()

Protected Member Functions

- BlBasicEffect (BlBasicEffect cloneSource)

  Creates a new BlBasicEffect by cloning parameter settings from an existing instance.
- override void OnApply ()

  Lazily computes derived parameter values immediately before applying the effect.

Properties

- Matrix World `[get, set]`

  Gets or sets the world matrix.
- Matrix View `[get, set]`

  Gets or sets the view matrix.
- Matrix Projection `[get, set]`

  Gets or sets the projection matrix.
- Vector3 DiffuseColor `[get, set]`

  Gets or sets the material diffuse color (range 0 to 1).
- Vector3 EmissiveColor `[get, set]`

  Gets or sets the material emissive color (range 0 to 1).
- Vector3 SpecularColor `[get, set]`

  Gets or sets the material specular color (range 0 to 1).
- float SpecularPower `[get, set]`

  Gets or sets the material specular power.
- float Alpha `[get, set]`

  Gets or sets the material alpha.
- bool LightingEnabled `[get, set]`
- bool PreferPerPixelLighting `[get, set]`

  Gets or sets the per-pixel lighting prefer flag.
- Vector3 AmbientLightColor `[get, set]`
- DirectionalLight DirectionalLight0 `[get]`
- DirectionalLight DirectionalLight1 `[get]`
- DirectionalLight DirectionalLight2 `[get]`
- bool FogEnabled `[get, set]`
- float FogStart `[get, set]`
- float FogEnd `[get, set]`
- Vector3 FogColor `[get, set]`
- bool TextureEnabled `[get, set]`

  Gets or sets whether texturing is enabled.
- Texture2D Texture `[get, set]`

  Gets or sets the current texture.
- bool VertexColorEnabled `[get, set]`

  Gets or sets whether vertex color is enabled.

6.1.1 Detailed Description

Holds a Blotch3D custom effect (like BlBasicEffectAlphaTest and BlBasicEffectClipColor) that is exactly like Mono↩ Game's BasicEffect but with certain added features. To make a BlBasicEffect, you must pass the mgfxo file name, or a byte array of its contents, to the constructor. See the SpriteAlphaTexture example and the section on Translucency for details on how to make and use objects of this class.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 BlBasicEffect() [1/3]

```
Blotch.BlBasicEffect.BlBasicEffect (
            GraphicsDevice device,
            string filename )
```

Creates a new BlBasicEffect with default parameter settings. See class description for more info.

6.1.2.2 BlBasicEffect() [2/3]

```
Blotch.BlBasicEffect.BlBasicEffect (
            GraphicsDevice device,
            byte [] bytes )
```

Creates a new BlBasicEffect with default parameter settings. See class description for more info.

6.1.2.3 BlBasicEffect() [3/3]

```
Blotch.BlBasicEffect.BlBasicEffect (
            BlBasicEffect cloneSource )  [protected]
```

Creates a new BlBasicEffect by cloning parameter settings from an existing instance.

6.1.3 Member Function Documentation

6.1.3.1 Clone()

```
override Effect Blotch.BlBasicEffect.Clone ( )
```

Creates a clone of the current BasicEffect instance.

6.1.3.2 OnApply()

```
override void Blotch.BlBasicEffect.OnApply ( )  [protected]
```

Lazily computes derived parameter values immediately before applying the effect.

6.1.4 Property Documentation

6.1.4.1 Alpha

```
float Blotch.BlBasicEffect.Alpha  [get], [set]
```

Gets or sets the material alpha.

6.1.4.2 DiffuseColor

```
Vector3 Blotch.BlBasicEffect.DiffuseColor  [get], [set]
```

Gets or sets the material diffuse color (range 0 to 1).

6.1.4.3 EmissiveColor

```
Vector3 Blotch.BlBasicEffect.EmissiveColor  [get], [set]
```

Gets or sets the material emissive color (range 0 to 1).

6.1.4.4 PreferPerPixelLighting

```
bool Blotch.BlBasicEffect.PreferPerPixelLighting  [get], [set]
```

Gets or sets the per-pixel lighting prefer flag.

6.1.4.5 Projection

```
Matrix Blotch.BlBasicEffect.Projection  [get], [set]
```

Gets or sets the projection matrix.

6.1.4.6 SpecularColor

```
Vector3 Blotch.BlBasicEffect.SpecularColor  [get], [set]
```

Gets or sets the material specular color (range 0 to 1).

6.1.4.7 SpecularPower

```
float Blotch.BlBasicEffect.SpecularPower  [get], [set]
```

Gets or sets the material specular power.

6.1.4.8 Texture

```
Texture2D Blotch.BlBasicEffect.Texture  [get], [set]
```

Gets or sets the current texture.

6.1.4.9 TextureEnabled

```
bool Blotch.BlBasicEffect.TextureEnabled  [get], [set]
```

Gets or sets whether texturing is enabled.

6.1.4.10 VertexColorEnabled

```
bool Blotch.BlBasicEffect.VertexColorEnabled  [get], [set]
```

Gets or sets whether vertex color is enabled.

6.1.4.11 View

```
Matrix Blotch.BlBasicEffect.View  [get], [set]
```

Gets or sets the view matrix.

6.1.4.12 World

```
Matrix Blotch.BlBasicEffect.World  [get], [set]
```

Gets or sets the world matrix.

The documentation for this class was generated from the following file:

- C:/Users/kloum/Desktop/Source/Blotch3D/src/BlBasicEffect.cs

6.2 Blotch.BIGeometry Class Reference

Methods and helpers for creating various geometric objects. These methods create and manage regular (rectangular) grids of vertices as a flattened row-major array (if indexed as [y, x]) of VertexPositionNormalTexture[], triangle arrays (also as a VertexPositionNormalTexture[]), and VertexBuffers. You can concatenate multiple regular grids to produce one regular grid if they have the same number of columns, and you can concatentate multiple triangle arrays to produce one triangle array. If you are confused by this, see the examples. You can transform either type of array with TransformVertices. You can create smooth normals for a regular grid and facet normals for a triangle array. You can set texture (UV) coordinates. You can convert a regular grid to a triangle array. Finally, you can convert a triangle array to a VertexBuffer suitable for adding to a BISprite.LODs field.

Public Member Functions

- delegate double XYToZDelegate (int x, int y)

  The delegate passed to certain geometry methods. Given an X and Y value, return a Z value.
- delegate Vector3 XYToVector3Delegate (int x, int y)

  The delegate passed to certain geometry methods. Given an X and Y value, return a Vector3. [Not yet used]

Static Public Member Functions

- static VertexPositionNormalTexture [] CreatePlanarSurface (Texture2D tex, bool mirrorY=false, bool smooth=true, double noiseLevel=Double.NaN, int numSignificantBits=8)

  Creates a square 1x1 surface in XY but with variation of its Z depending on the pixels in an image (heightfield). A maximum pixel value (according to numSignificantBits) causes the corresponding position on the surface to have a height of 1. Use TransformVertices to alter this. Returns a triangle array of the surface, which includes smooth normals and texture coordinates.
- static VertexPositionNormalTexture [] CreatePlanarSurface (XYToZDelegate pixelFunc, int numX=256, int numY=256, bool mirrorY=false, bool smooth=false, double noiseLevel=0)

  Creates a square 1x1 surface in XY but with variation of its Z depending on the result of a delegate. Returns a triangle array of the surface, which includes smooth normals and texture coordinates.
- static VertexPositionNormalTexture [] CreatePlanarSurface (double[,] heightMap, bool mirrorY=false, bool smooth=false, double noiseLevel=0)

  Creates a square 1x1 surface in XY but with variation of its Z depending on the elements of a 2D array of doubles. Returns a triangle array of the surface, which includes smooth normals and texture coordinates.
- static VertexPositionNormalTexture [] CreateCylindroid (XYToZDelegate pixelFunc, int numHorizVertices=32, int numVertVertices=2, double topDiameter=1, bool facetedNormals=false, bool createBody=true, bool createEndCaps=true, Matrix? matrix=null)

  Like the CreateCylindroidSurface overload that takes a heightMap (see that method for details), but rather than a heightMap, this takes a delegate that defines the diameter multiplier.
- static VertexPositionNormalTexture [] CreateCylindroid (int numHorizVertices=32, int numVertVertices=2, double topDiameter=1, bool facetedNormals=false, double[,] heightMap=null, bool createBody=true, bool createEndCaps=true, Matrix? matrix=null)

  Creates a cylindroid (including texture coords and normals) and/or the end caps of the cylindroid, with the given parameters and returns a triangle array, which includes smooth normals and texture coordinates. Assuming a possible subsequent call to TransformVertices, even without a heightMap many fundamental rotationally symmetric shapes can be generated, like a cylinder, cone, washer, disk, prism of any number of facets, tetrahedron, pyramid of any number of facets, etc. Before passing the result to TransformVertices, the center of the cylindroid is the origin, its height is 1, the diameter of the base is 1, and the diameter of the top is topDiameter. If heightMap is specified, each element multiplies the parameterized diameter at the corresponding point on the surface. The dimensions of heightMap can be different from the dimensions of the cylindroid. (Note that in C#, the second index of a 2D array is the rows. So heightMap array indices must be of the form [y, x].) heightMap is mapped onto the object such that the heightMap X wraps around horizontally and the heightMap Y is mapped vertically to the height (Z) of the object. For example, if the heightMap X dimension is 1, then it defines the diameter shape that is rotated around the whole cylindroid. For some shapes you may also want to re-calculate normals with CalcFacetNormals (for example, if the the subsequent transform caused some normals to become invalid). You can create the body and endcaps separately so they can be assigned to different sprites and thus have different textures, or create them together. See the GeomObjectsWith↩ Heightmap example.

- static VertexPositionNormalTexture [ ] CalcCylindroidVerticesAndTexcoords (int numX=32, int numY=2, double topDiameter=1, double[,] heightMap=null, Matrix? matrix=null)

  Like #CreateCylindroidSurface, but returns a row-major regular grid rather than a triangle list, and doesn't calculate the normals, so you'll need to do that separately with the appropriate functions.
- static VertexPositionNormalTexture [ ] VerticesToTriangles (VertexPositionNormalTexture[ ] vertices, int numX)

  Given a row-major [y, x] regular grid of vertices, return an array of triangles. numY is assumed to be the length of vertices/numX.
- static VertexBuffer TrianglesToVertexBuffer (GraphicsDevice graphicsDevice, VertexPositionNormalTexture[ ] vertices)

  Given a triangle list, returns a VertexBuffer
- static VertexPositionNormalTexture [ ] TransformVertices (VertexPositionNormalTexture[ ] vertices, Matrix matrix)

  Transforms a regular grid or triangle array (including transforming the transpose of the inverse of each normal) according to the specified matrix. If it's an array of triangles, you will probably want to call CullEmptyTriangles if the transform might cause some triangles to have zero area, and maybe CalcSmoothNormals (for regular grids) or CalcFacetNormals (for triangles) afterward if the transform might cause normals to be invalid or point the wrong way, causing the surface to be black or the wrong brightness (typically when a dimenson is scaled to zero or inverted).
- static BoundingSphere GetBoundingSphere (VertexPositionNormalTexture[ ] vertices)

  Returns the BoundingSphere for the specified vertices or triangles
- static VertexPositionNormalTexture [ ] CullEmptyTriangles (VertexPositionNormalTexture[ ] triangles)

  Removes triangles that have zero area. Typically called after a transform.
- static VertexPositionNormalTexture [ ] UnsmoothEdgeNormals (VertexPositionNormalTexture[ ] triangles, double thresholdAngle=.2)

  If any triangles in the triangle array have vertex normals that vary in direction more than the specified angular value (thresholdAngle), then set all the normals in that triangle to facet normals (i.e. perpendicular to the triangle).
- static VertexPositionNormalTexture [ ] CalcSmoothNormals (VertexPositionNormalTexture[ ] vertices, int numX, bool xIsWrapped=false, bool invert=false)

  For a row-major (indexed as [y, x]) regular grid (i.e. NOT triangles), calculates a normal for each point in the grid. The normal for a given point is an average of the normals of the (typically eight) triangles that the vertex would participate in. (Of course, the triangles have not yet been separated-out.) numY is assumed to be vertices.Length/numX.
- static VertexPositionNormalTexture [ ] ScaleNormals (VertexPositionNormalTexture[ ] vertices, double scale=-1)

  Scales the normals.
- static VertexPositionNormalTexture [ ] SetTextureToXY (VertexPositionNormalTexture[ ] vertices)

  Set texture coordinates (UV) to normalized XY planar.
- static VertexPositionNormalTexture [ ] CalcFacetNormals (VertexPositionNormalTexture[ ] vertices)

  Calculates one normal for each triangle in an existing 2D array of triangles (NOT a regular grid of points). The normal for each triangle is orthogonal to its surface.
- static VertexPositionNormalTexture [ ] ReverseTriangles (VertexPositionNormalTexture[ ] vertices)

  Reverses all the triangles in a triangle array
- static VertexPositionNormalTexture [ ] CalcPlanarVerticesAndTexcoords (double[,] heightMap, double noiseLevel=0, bool mirrorY=false, bool smooth=false)

  Calculate vertices and texture coordinates, but not normals, from a specified row-major (indexed as [y, x]) heightmap double array. Returns a 1x1 surface in XY, but with Z for a given position equal to the corresponding heightMap element.

6.2.1 Detailed Description

Methods and helpers for creating various geometric objects. These methods create and manage regular (rectangular) grids of vertices as a flattened row-major array (if indexed as [y, x]) of VertexPositionNormalTexture[], triangle arrays (also as a VertexPositionNormalTexture[]), and VertexBuffers. You can concatenate multiple regular grids to produce one regular grid if they have the same number of columns, and you can concatentate multiple triangle arrays to produce one triangle array. If you are confused by this, see the examples. You can transform either type of array with TransformVertices. You can create smooth normals for a regular grid and facet normals for a triangle array. You can set texture (UV) coordinates. You can convert a regular grid to a triangle array. Finally, you can convert a triangle array to a VertexBuffer suitable for adding to a BISprite.LODs field.

6.2.2 Member Function Documentation

6.2.2.1 CalcCylindroidVerticesAndTexcoords()

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.CalcCylindroidVerticesAndTexcoords (
            int numX = 32,
            int numY = 2,
            double topDiameter = 1,
            double heightMap[,] = null,
            Matrix?  matrix = null )  [static]
```

Like #CreateCylindroidSurface, but returns a row-major regular grid rather than a triangle list, and doesn't calculate the normals, so you'll need to do that separately with the appropriate functions.

Parameters

numX	The number of X elements in a row
numY	The number of Y elements in a column
topDiameter	Diameter of top of cylindroid (if heightMap==null)
heightMap	See CreateCylindroidSurface
matrix	Matrix to transform each increment of y level from previous y level

Returns

A list of the cylindroid's vertices

6.2.2.2 CalcFacetNormals()

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.CalcFacetNormals (
            VertexPositionNormalTexture [] vertices )  [static]
```

Calculates one normal for each triangle in an existing 2D array of triangles (NOT a regular grid of points). The normal for each triangle is orthogonal to its surface.

Parameters

vertices	A flattened (in row-major order) 2D array of triangles (this array is changed to be the output array)

Returns

Array with normals calculated

6.2.2.3 CalcPlanarVerticesAndTexcoords()

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.CalcPlanarVerticesAndTexcoords (
            double heightMap[,],
            double noiseLevel = 0,
            bool mirrorY = false,
            bool smooth = false )  [static]
```

Calculate vertices and texture coordinates, but not normals, from a specified row-major (indexed as [y, x]) heightmap double array. Returns a 1x1 surface in XY, but with Z for a given position equal to the corresponding heightMap element.

Parameters

heightMap	A row-major array (indexed as [y, x]) of height values
noiseLevel	How much noise to add
mirrorY	Whether to invert the Y dimension
smooth	Whether to apply a 3x3 gaussian blur on each pixel height

6.2.2.4 CalcSmoothNormals()

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.CalcSmoothNormals (
            VertexPositionNormalTexture [] vertices,
            int numX,
            bool xIsWrapped = false,
            bool invert = false )  [static]
```

For a row-major (indexed as [y, x]) regular grid (i.e. NOT triangles), calculates a normal for each point in the grid. The normal for a given point is an average of the normals of the (typically eight) triangles that the vertex would participate in. (Of course, the triangles have not yet been separated-out.) numY is assumed to be vertices.Length/numX.

Parameters

vertices	A flattened (in row-major order) 2D array of vertices (this method may change the contents of this grid)
numX	The number of X elements in a row
xIsWrapped	Include the row-wrapped ponts in the calculation of normals on the row edge. Closed cylindroids where x is wrapped would need this.
invert	Inverts the normals (typically when viewing faces from the inside)

Returns

The input grid with smooth normals added

6.2.2.5 CreateCylindroid() [1/2]

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.CreateCylindroid (
            XYToZDelegate pixelFunc,
```

```
            int numHorizVertices = 32,
            int numVertVertices = 2,
            double topDiameter = 1,
            bool facetedNormals = false,
            bool createBody = true,
            bool createEndCaps = true,
            Matrix?  matrix = null )  [static]
```

Like the CreateCylindroidSurface overload that takes a heightMap (see that method for details), but rather than a heightMap, this takes a delegate that defines the diameter multiplier.

Parameters

pixelFunc	A delegate that takes an x and y and returns the diameter multiplier
numHorizVertices	The number of horizontal vertices in a row
numVertVertices	The number of vertical vertices in a column
topDiameter	Diameter of top of cylindroid (if heightMap==null)
facetedNormals	If true, create normals per triangle. If false, create smooth normals
createBody	Whether to create the body of the cylindroid
createEndCaps	Whether to create a cap for each end
matrix	Matrix to transform each increment of y level from previous y level

Returns

6.2.2.6 CreateCylindroid() [2/2]

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.CreateCylindroid (
            int numHorizVertices = 32,
            int numVertVertices = 2,
            double topDiameter = 1,
            bool facetedNormals = false,
            double heightMap[,] = null,
            bool createBody = true,
            bool createEndCaps = true,
            Matrix?  matrix = null )  [static]
```

Creates a cylindroid (including texture coords and normals) and/or the end caps of the cylindroid, with the given parameters and returns a triangle array, which includes smooth normals and texture coordinates. Assuming a possible subsequent call to TransformVertices, even without a heightMap many fundamental rotationally symmetric shapes can be generated, like a cylinder, cone, washer, disk, prism of any number of facets, tetrahedron, pyramid of any number of facets, etc. Before passing the result to TransformVertices, the center of the cylindroid is the origin, its height is 1, the diameter of the base is 1, and the diameter of the top is topDiameter. If heightMap is specified, each element multiplies the parameterized diameter at the corresponding point on the surface. The dimensions of heightMap can be different from the dimensions of the cylindroid. (Note that in C#, the second index of a 2D array is the rows. So heightMap array indices must be of the form [y, x].) heightMap is mapped onto the object such that the heightMap X wraps around horizontally and the heightMap Y is mapped vertically to the height (Z) of the object. For example, if the heightMap X dimension is 1, then it defines the diameter shape that is rotated around the whole cylindroid. For some shapes you may also want to re-calculate normals with CalcFacetNormals (for example, if the the subsequent transform caused some normals to become invalid). You can create the body and endcaps separately so they can be assigned to different sprites and thus have different textures, or create them together. See the GeomObjectsWithHeightmap example.

Parameters

numHorizVertices	The number of horizontal vertices in a row
numVertVertices	The number of vertical vertices in a column
topDiameter	Diameter of top of cylindroid (if heightMap==null)
facetedNormals	If true, create normals per triangle. If false, create smooth normals
heightMap	If not null, then this is mapped onto the surface to modify the diameter. See method description for details, but note that indices must be of the form [y, x]. This need not have the same dimensions as the cylindroid.
createBody	Whether to create the body
createEndCaps	Whether to create a cap for each end
matrix	Matrix to transform each increment of y level from previous y level

Returns

A triangle list of the cylindroid

6.2.2.7 CreatePlanarSurface() [1/3]

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.CreatePlanarSurface (
            Texture2D tex,
            bool mirrorY = false,
            bool smooth = true,
            double noiseLevel = Double.NaN,
            int numSignificantBits = 8 )  [static]
```

Creates a square 1x1 surface in XY but with variation of its Z depending on the pixels in an image (heightfield). A maximum pixel value (according to numSignificantBits) causes the corresponding position on the surface to have a height of 1. Use TransformVertices to alter this. Returns a triangle array of the surface, which includes smooth normals and texture coordinates.

Parameters

tex	The texture that represents the height (Z) of each vertex.
mirrorY	If true, then invert image's Y dimension
smooth	Whether to apply a 3x3 gaussian smoothing kernel, or not
noiseLevel	How much noise to add. If it's Double.NaN, then automatically calculate a little noise to overcome quantization from limited numSignificantBits
numSignificantBits	How many bits in a pixel should be used (starting from the least significant bit). Normally the first 8 bits are used (the last channel), but special images might combine the bits of multiple channels.

Returns

The triangles of a terrain from the specified image, including smooth normals and texture coordinates

6.2.2.8 CreatePlanarSurface() [2/3]

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.CreatePlanarSurface (
            XYToZDelegate pixelFunc,
            int numX = 256,
            int numY = 256,
            bool mirrorY = false,
            bool smooth = false,
            double noiseLevel = 0 )  [static]
```

Creates a square 1x1 surface in XY but with variation of its Z depending on the result of a delegate. Returns a triangle array of the surface, which includes smooth normals and texture coordinates.

Parameters

pixelFunc	A delegate that takes the x and y and returns that pixel's height
numX	The number of X elements in a row
numY	The number of X elements in a row
mirrorY	Whether to invert Y
smooth	Whether to apply a 3x3 gaussian smoothing kernel, or not
noiseLevel	How much noise to add

Returns

Triangles of the surface, including smooth normals and teture coordinates

6.2.2.9 CreatePlanarSurface() [3/3]

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.CreatePlanarSurface (
            double heightMap[,],
            bool mirrorY = false,
            bool smooth = false,
            double noiseLevel = 0 )  [static]
```

Creates a square 1x1 surface in XY but with variation of its Z depending on the elements of a 2D array of doubles. Returns a triangle array of the surface, which includes smooth normals and texture coordinates.

Parameters

heightMap	A row-major (must be indexed as [y, x]) array of vertex heights.
mirrorY	Whether to invert Y
smooth	Whether to apply a 3x3 gaussian smoothing kernel, or not
noiseLevel	How much noise to add

Returns

Triangles of the surface, including smooth normals and teture coordinates

6.2.2.10 CullEmptyTriangles()

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.CullEmptyTriangles (
            VertexPositionNormalTexture [] triangles )  [static]
```

Removes triangles that have zero area. Typically called after a transform.

Parameters

triangles	The input triangles (i.e. NOT a regular grid)

Returns

Output triangles

6.2.2.11 GetBoundingSphere()

```
static BoundingSphere Blotch.BlGeometry.GetBoundingSphere (
            VertexPositionNormalTexture [] vertices )  [static]
```

Returns the BoundingSphere for the specified vertices or triangles

Parameters

vertices	Vertices or triangles to get the BoundingSphere for

Returns

The BoundingSphere for the specified vertices or triangles

6.2.2.12 ReverseTriangles()

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.ReverseTriangles (
            VertexPositionNormalTexture [] vertices )  [static]
```

Reverses all the triangles in a triangle array

Parameters

vertices	The triangle array to reverse, and also altered to be the return array

Returns

The triangle array with its triangles reversed

6.2.2.13 ScaleNormals()

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.ScaleNormals (
            VertexPositionNormalTexture [] vertices,
            double scale = -1 )  [static]
```

Scales the normals.

Parameters

vertices	Input array of vertices, and output array as well
scale	Scales applied to each normal

6.2.2.14 SetTextureToXY()

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.SetTextureToXY (
            VertexPositionNormalTexture [] vertices )  [static]
```

Set texture coordinates (UV) to normalized XY planar.

Parameters

vertices	Input array of vertices, and output array as well

6.2.2.15 TransformVertices()

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.TransformVertices (
            VertexPositionNormalTexture [] vertices,
            Matrix matrix )  [static]
```

Transforms a regular grid or triangle array (including transforming the transpose of the inverse of each normal) according to the specified matrix. If it's an array of triangles, you will probably want to call CullEmptyTriangles if the transform might cause some triangles to have zero area, and maybe CalcSmoothNormals (for regular grids) or CalcFacetNormals (for triangles) afterward if the transform might cause normals to be invalid or point the wrong way, causing the surface to be black or the wrong brightness (typically when a dimenson is scaled to zero or inverted).

Parameters

vertices	Input array (this is altered by the method)
matrix	Transformation matrix

Returns

The transformed array

6.2.2.16 TrianglesToVertexBuffer()

```
static VertexBuffer Blotch.BlGeometry.TrianglesToVertexBuffer (
            GraphicsDevice graphicsDevice,
            VertexPositionNormalTexture [] vertices )  [static]
```

Given a triangle list, returns a VertexBuffer

Parameters

graphicsDevice	The graphics device to use
vertices	The triangles to convert to a VertexBuffer

Returns

The VertexBuffer that contains the triangle list

6.2.2.17 UnsmoothEdgeNormals()

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.UnsmoothEdgeNormals (
            VertexPositionNormalTexture [] triangles,
            double thresholdAngle = .2 )  [static]
```

If any triangles in the triangle array have vertex normals that vary in direction more than the specified angular value (thresholdAngle), then set all the normals in that triangle to facet normals (i.e. perpendicular to the triangle).

Parameters

triangles	The input (and output) triangles (i.e. NOT a regular grid)
thresholdAngle	The angle, in radians, that normals must vary in a given triangle to merit setting facet normals for that triangle

Returns

The output (and input) triangles, altered to reflect this unsmooth function

6.2.2.18 VerticesToTriangles()

```
static VertexPositionNormalTexture [] Blotch.BlGeometry.VerticesToTriangles (
            VertexPositionNormalTexture [] vertices,
            int numX )  [static]
```

Given a row-major [y, x] regular grid of vertices, return an array of triangles. numY is assumed to be the length of vertices/numX.

Parameters

vertices	A flattened row-major array of points
numX	The number of X elements in a row

Returns

Triangle array

6.2.2.19 XYToVector3Delegate()

```
delegate Vector3 Blotch.BlGeometry.XYToVector3Delegate (
            int x,
            int y )
```

The delegate passed to certain geometry methods. Given an X and Y value, return a Vector3. [Not yet used]

Parameters

x	The x of the surface position
y	The y of the surface position

Returns

The position of the surface for the corresponding vertex grid XY

6.2.2.20 XYToZDelegate()

```
delegate double Blotch.BlGeometry.XYToZDelegate (
            int x,
            int y )
```

The delegate passed to certain geometry methods. Given an X and Y value, return a Z value.

Parameters

x	The x of the surface position
y	The y of the surface position

Returns

The height or diameter multiplier of the surface at the corresponding XY position

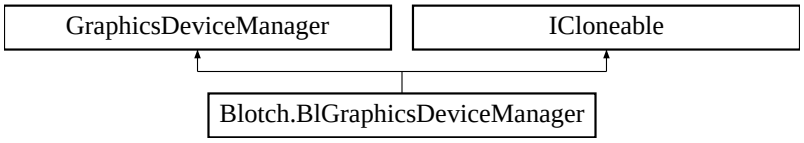
The documentation for this class was generated from the following file:

- C:/Users/kloum/Desktop/Source/Blotch3D/src/BlGeometry.cs

6.3 Blotch.BlGraphicsDeviceManager Class Reference

This holds everything having to do with an output device. BlWindow3D creates one of these for itself. This derives from MonoGame GraphicsDeviceManager.

Inheritance diagram for Blotch.BlGraphicsDeviceManager:



Classes

- class Light

  Defines a light. See the Lights field. The default BasicShader supports up to three lights.

Public Member Functions

- BlGraphicsDeviceManager (BlWindow3D window)
- void Initialize ()

  For internal use only. Apps should not normally call this. This initializes some values AFTER the BlWindow3D has been created.
- void ExtendClippingTo (BlSprite s)

  Informs the auto-clipping code of an object that should be visible within the clipping limits. This is mainly for internal use. Application code should control clipping with NearClip and FarClip.
- void SetSpriteToCamera (BlSprite sprite)

  Sets a sprite's BlSprite::Matrix to the current camera position and orientation. You could use this to implement a HUD, for example. Note: This only works correctly if the sprite has no parent (and is thus drawn directly) or it's parents are untransformed. If all you want is to set the sprite's position (but NOT orientation) to the camera, then set the sprite's Matrix.Translation = graphics.Eye
- void SetCameraToSprite (BlSprite sprite)

  Sets the camera position and orientation to the current position and orientation of a sprite. You could use for cockpit view, for example. Note that the camera will lag sprite movement unless the following is done: For every frame you must first calculate the sprite's position and orientation, call this function, and then draw everything.
- void AdjustCameraZoom (double dif)

  Sets the Zoom. If dif is zero, then there is no change in zoom. Normally one would set zoom with the Zoom field. This is mainly for internal use.
- void AdjustCameraDolly (double dif)

  Migrates the current camera dolly (distance from LookAt) according to dif. If dif is zero, then there is no change in dolly.

- void AdjustCameraTruck (double difX, double difY=0)

  Adjusts camera truck (movement relative to camera direction) according to difX and difY. if difX and difY are zero, then truck position isn't changed.
- void AdjustCameraRotation (double difX, double difY=0)

  Adjusts camera rotation about the LookAt point according to difX and difY. if difX and difY are zero, then rotation isn't changed.
- void AdjustCameraPan (double difX, double difY=0)

  Adjusts camera pan (changing direction of camera) according to difX and difY. if difX and difY are zero, then pan direction isn't changed.
- Ray DoDefaultGui ()

  Updates Eye, LookAt, etc. according to mouse and certain key input. Specifically: Wheel=Dolly, CTRL-wheel=Zoom, Left-drag=Truck, Right-drag=Rotate, CTRL-left-drag=Pan, Esc=Reset. Also, SHIFT causes all the previous controls to be fine rather than coarse. If CTRL is pressed and mouse left or right button is clicked, then returns a ray into window at mouse position. To control each camera attribute individually and programatically or override the GUI controls, use AdjustCameraZoom, AdjustCameraDolly, AdjustCameraRotation, AdjustCameraPan, AdjustCamera↩ Truck, ResetCamera, and/or SetCameraToSprite. Or see the more basic fields of Zoom, Aspect, TargetEye, and TargetLookAt.
- void ResetCamera ()

  Sets Eye. LookAt, etc. back to default starting position.
- void SetCameraRollToZero ()

  Sets the camera 'roll' to be level with the XY plane
- Ray CalculateRay (Vector2 windowPosition)

  Returns a ray that that goes from the near clipping plane to the far clipping plane, at the specified window position.
- Vector3 GetWindowCoordinates (BlSprite sprite)

  Returns the window coordinates of the specified sprite.
- Texture2D TextToTexture (string text, SpriteFont font, Microsoft.Xna.Framework.Color? color=null, Microsoft.Xna.Framework.Color? backColor=null)

  Returns a Texture2D containing the specified text. It's up to the caller to Dispose the returned texture.
- void DrawTexture (Texture2D texture, Rectangle windowRect, Microsoft.Xna.Framework.Color? color=null)

  Draws a texture in the window.
- void DrawText (string text, SpriteFont font, Vector2 windowPos, Microsoft.Xna.Framework.Color? color=null)

  Draws text on the window.
- Texture2D LoadFromImageFile (string fileName, bool mirrorY=false)

  Loads a texture directly from an image file.
- void PrepareDraw (bool firstCallInDraw=true)

  This is automatically called once at the beginning of your BlWindow3D::FrameDraw method. It calculates the latest View and Projection settings according to the current camera specifications (Zoom, Aspect, Eye, LookAt, etc.), and if firstCallInDraw is true it also may sleep in order to obey FramePeriod. It must also be called explicitly after any changes to the camera settings made later in the BlWindow3D::FrameDraw method. Only in the first call should firstCallInDraw be true, and in any subsequent calls it should be false.
- Texture2D CloneTexture2D (Texture2D tex)

  Returns a deepcopy of the texture
- object **Clone** ()
- new void Dispose ()

  When finished with the object, you must call Dispose() from the same thread that created the object. You can call this multiple times, but once is enough.

Public Attributes

- Microsoft.Xna.Framework.Matrix View

  This is the view matrix. Normally you would use the higher-level functions Eye, LookAt, CameraUp, SetCameraToSprite, and DoDefaultGui intead of changing this directly.
- Microsoft.Xna.Framework.Matrix Projection

The Projection matrix. Normally you would use the higher-level functions Zoom, Aspect, NearClip, or FarClip intead of changing this directly.

- Vector3 CameraUp

  Camera Up vector. Initially set to +Z. ResetCamera and SetCameraToSprite updates this.

- double DefGuiMinLookZ = -1

  Causes DoDefaultGui to prevent the Z component of CameraForwardNormalized from falling below this value. For example, set this to zero so that DoDefaultGui won't allow the camera to look downward

- double DefGuiMaxLookZ = 1

  Caues DoDefaultGui to prevent the Z component of CameraForwardNormalized from rising above this value. For example, set this to zero so that DoDefaultGui won't allow the camera to look upward

- DepthStencilState DepthStencilStateEnabled

  Assign DepthStencilState to this to enable depth buffering

- DepthStencilState DepthStencilStateDisabled

  Assign DepthStencilState to this to disable depth buffering

- Vector3 TargetEye

  The point that Eye migrates to, according to CameraSpeed. This is normally controlled by DoDefaultGui, but can also be controlled by the AdjustCameraxxx methods. The easiest way to control the camera exactly (including camera roll), is to use SetCameraToSprite and then set the sprite's matrix as desired.

- Vector3 TargetLookAt

  The point that LookAt migrates to, according to CameraSpeed. This is normally controlled by DoDefaultGui, but can also be controlled by the AdjustCameraxxx methods. The easiest way to control the camera exactly (including camera roll), is to use SetCameraToSprite and then set the sprite's matrix as desired.

- double CameraSpeed = .4

  The responsiveness of the camera position to changes in TargetEye and TargetLookAt. A value of 0 means it doesn't respond to changes, 1 means it immediately responds. See Eye and LookAt for more information.

- double Zoom =45

  The field of view, in degrees.

- double Aspect =2

  The aspect ratio.

- double NearClip = 0

  The near clipping plane. If 0 then autoclip. If negative then auto-clip down to a limit of -NearClip. (Auto-clipping has a one-frame latency. A mechanism is employed in the camera control methods to somewhat alleviate this, but in certain cases you may still see a one frame dropout in visibility.) See the description for the depth buffer for more information.

- double FarClip = 0

  The far clipping plane. If 0 then autoclip. If negative then auto-clip up to a limit of -FarClip. (Auto-clipping has a one-frame latency. A mechanism is employed in the camera control methods to somewhat alleviate this, but in certain cases you may still see a one frame dropout in visibility.) See the description for the depth buffer for more information.

- double ClipRangeExcess = 5

  Increase far clipping and decrease near clipping by this much. Use this to alleviate certain auto-clipping artifacts. See NearClip and FarClip.

- Microsoft.Xna.Framework.Color ClearColor =new Microsoft.Xna.Framework.Color(0,0,.1f)

  The background color.

- double AutoRotate = 0

  How fast DoDefaultGui should auto-rotate the scene.

- double FramePeriod = 1/60.0

  How much time between consecutive frames.

- List< Light > Lights = new List<Light>()

  The directional lights. Note: The BasicEffect shader only supports the first three. To handle more lights, you'll need to write your own shader.

- Vector3 AmbientLightColor = new Vector3(.1f, .1f, .1f)

  The ambient light color. If null, no ambient light is enabled. Note: There is no ambient color for a BISprite. Both diffuse and ambient light illuminates the model's Color. See the BISprite::Color member.

- Vector3 FogColor = null

  If not null, color of fog.

- float fogStart = 1

  How far away fog starts. See FogColor.
- float fogEnd = 10

  How far away fog ends. See FogColor.
- BlWindow3D Window

  The BlWindow3D associated with this object.
- SpriteBatch SpriteBatch =null

  A SpriteBatch for use by certain text and texture drawing methods.
- bool IsDisposed = false

  Set when the object is Disposed.

Properties

- Vector3 CameraForward `[get]`

  The vector between Eye and LookAt. Writes to Eye and LookAt and calls to SetCameraToSprite cause this to be updated. Also see CameraForwardNormalized and CameraForwardMag.
- Vector3 CameraForwardNormalized `[get]`

  Normalized form of CameraForward. Writes to Eye and LookAt, and calls to SetCameraToSprite cause this to be updated. Also see CameraForward and CameraForwardMag.
- float CameraForwardMag `[get]`

  The magnitude of CameraForward. Writes to Eye and LookAt, and calls to SetCameraToSprite cause this to be updated. Also see CameraForward and CameraForwardNormalized.
- Vector3 CameraRight `[get]`

  Camera Right vector. Writes to Eye and LookAt, and calls to SetCameraToSprite cause this to be updated.
- Vector3 Eye `[get]`

  The current camera position. See TargetEye.
- Vector3 LookAt `[get]`

  The current camera LookAt position. See TargetLookAt.
- double CurrentAspect `[get]`

  Current aspect ratio. Same as Aspect unless Aspect==0.
- double CurrentNearClip `[get]`

  Current value of near clipping plane. See NearClip.
- double CurrentFarClip `[get]`

  Current value of far clipping plane. See FarClip.
- double MinCamDistance `[get]`

  Distance to the nearest sprite, less its radius. Note this is set to a very large number by PrepareDraw, and then as BlWindow3D::FrameDraw is called it is set more reasonably.
- double MaxCamDistance `[get]`

  Distance to the farthest sprite, plus its radius. Note this is set to a very small number by PrepareDraw, and then as BlWindow3D::FrameDraw is called it is set more reasonably.

6.3.1 Detailed Description

This holds everything having to do with an output device. BlWindow3D creates one of these for itself. This derives from MonoGame GraphicsDeviceManager.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 BlGraphicsDeviceManager()

```
Blotch.BlGraphicsDeviceManager.BlGraphicsDeviceManager (
            BlWindow3D window )
```

Parameters

window	The BlWindow3D object for which this is to be the BlGraphicsDeviceManager

6.3.3 Member Function Documentation

6.3.3.1 AdjustCameraDolly()

```
void Blotch.BlGraphicsDeviceManager.AdjustCameraDolly (
            double dif )
```

Migrates the current camera dolly (distance from LookAt) according to dif. If dif is zero, then there is no change in dolly.

Parameters

dif	How much to dolly camera (plus = toward LookAt, minus = away)

6.3.3.2 AdjustCameraPan()

```
void Blotch.BlGraphicsDeviceManager.AdjustCameraPan (
            double difX,
            double difY = 0 )
```

Adjusts camera pan (changing direction of camera) according to difX and difY. if difX and difY are zero, then pan direction isn't changed.

Parameters

difX	How much to pan horizontally
difY	How much to pan vertically

6.3.3.3 AdjustCameraRotation()

```
void Blotch.BlGraphicsDeviceManager.AdjustCameraRotation (
            double difX,
            double difY = 0 )
```

Adjusts camera rotation about the LookAt point according to difX and difY. if difX and difY are zero, then rotation isn't changed.

Parameters

difX	How much to rotate the camera horizontally
difY	How much to rotate the camera vertically

6.3.3.4 AdjustCameraTruck()

```
void Blotch.BlGraphicsDeviceManager.AdjustCameraTruck (
            double difX,
            double difY = 0 )
```

Adjusts camera truck (movement relative to camera direction) according to difX and difY. if difX and difY are zero, then truck position isn't changed.

Parameters

difX	How much to truck the camera horizontally
difY	How much to truck the camera vertically

6.3.3.5 AdjustCameraZoom()

```
void Blotch.BlGraphicsDeviceManager.AdjustCameraZoom (
            double dif )
```

Sets the Zoom. If dif is zero, then there is no change in zoom. Normally one would set zoom with the Zoom field. This is mainly for internal use.

Parameters

dif	How much to zoom camera (plus = magnify, minus = reduce)

6.3.3.6 CalculateRay()

```
Ray Blotch.BlGraphicsDeviceManager.CalculateRay (
            Vector2 windowPosition )
```

Returns a ray that that goes from the near clipping plane to the far clipping plane, at the specified window position.

Parameters

windowPosition	The window's pixel coordinates

Returns

The Ray into the window at the specified pixel coordinates

6.3.3.7 CloneTexture2D()

```
Texture2D Blotch.BlGraphicsDeviceManager.CloneTexture2D (
            Texture2D tex )
```

Returns a deepcopy of the texture

Parameters

tex	The texture to deepcopy

Returns

A deepcopy of tex

6.3.3.8 Dispose()

```
new void Blotch.BlGraphicsDeviceManager.Dispose ( )
```

When finished with the object, you must call Dispose() from the same thread that created the object. You can call this multiple times, but once is enough.

6.3.3.9 DoDefaultGui()

```
Ray Blotch.BlGraphicsDeviceManager.DoDefaultGui ( )
```

Updates Eye, LookAt, etc. according to mouse and certain key input. Specifically: Wheel=Dolly, CTR↩ L-wheel=Zoom, Left-drag=Truck, Right-drag=Rotate, CTRL-left-drag=Pan, Esc=Reset. Also, SHIFT causes all the previous controls to be fine rather than coarse. If CTRL is pressed and mouse left or right button is clicked, then returns a ray into window at mouse position. To control each camera attribute individually and programatically or override the GUI controls, use AdjustCameraZoom, AdjustCameraDolly, AdjustCameraRotation, AdjustCamera↩ Pan, AdjustCameraTruck, ResetCamera, and/or SetCameraToSprite. Or see the more basic fields of Zoom, Aspect, TargetEye, and TargetLookAt.

Returns

If a mouse left or right click occurred, returns the Ray into the screen at that position. Otherwise returns null

6.3.3.10 DrawText()

```
void Blotch.BlGraphicsDeviceManager.DrawText (
            string text,
            SpriteFont font,
            Vector2 windowPos,
            Microsoft.Xna.Framework.Color?  color = null )
```

Draws text on the window.

Parameters

text	The text to draw
font	The font to use (typically created from SpriteFont content with Content.Load<SpriteFont>(...))
windowPos	The X and Y window location, in pixels
color	Foreground color of the font

6.3.3.11 DrawTexture()

```
void Blotch.BlGraphicsDeviceManager.DrawTexture (
            Texture2D texture,
            Rectangle windowRect,
            Microsoft.Xna.Framework.Color?  color = null )
```

Draws a texture in the window.

Parameters

texture	The texture to draw
windowRect	The X and Y window location, in pixels
color	Foreground color of the font

6.3.3.12 ExtendClippingTo()

```
void Blotch.BlGraphicsDeviceManager.ExtendClippingTo (
            BlSprite s )
```

Informs the auto-clipping code of an object that should be visible within the clipping limits. This is mainly for internal use. Application code should control clipping with NearClip and FarClip.

Parameters

s	The sprite that should be included in the auto-clipping code

6.3.3.13 GetWindowCoordinates()

```
Vector3 Blotch.BlGraphicsDeviceManager.GetWindowCoordinates (
            BlSprite sprite )
```

Returns the window coordinates of the specified sprite.

Parameters

sprite	The sprite to get the window coordinates of

Returns

The window coordinates of the sprite, in pixels

6.3.3.14 Initialize()

```
void Blotch.BlGraphicsDeviceManager.Initialize ( )
```

For internal use only. Apps should not normally call this. This initializes some values AFTER the BlWindow3D has been created.

6.3.3.15 LoadFromImageFile()

```
Texture2D Blotch.BlGraphicsDeviceManager.LoadFromImageFile (
            string fileName,
            bool mirrorY = false )
```

Loads a texture directly from an image file.

Parameters

fileName	An image file of any standard type supported by MonoGame (jpg, png, etc.)
mirrorY	If true, then mirror Y

Returns

The texture that was loaded

6.3.3.16 PrepareDraw()

```
void Blotch.BlGraphicsDeviceManager.PrepareDraw (
            bool firstCallInDraw = true )
```

This is automatically called once at the beginning of your BlWindow3D::FrameDraw method. It calculates the latest View and Projection settings according to the current camera specifications (Zoom, Aspect, Eye, LookAt, etc.), and if firstCallInDraw is true it also may sleep in order to obey FramePeriod. It must also be called explicitly after any changes to the camera settings made later in the BlWindow3D::FrameDraw method. Only in the first call should firstCallInDraw be true, and in any subsequent calls it should be false.

Parameters

firstCallInDraw	True indicates this method should also sleep in order to obey FramePeriod.

6.3.3.17 ResetCamera()

```
void Blotch.BlGraphicsDeviceManager.ResetCamera ( )
```

Sets Eye. LookAt, etc. back to default starting position.

6.3.3.18 SetCameraRollToZero()

```
void Blotch.BlGraphicsDeviceManager.SetCameraRollToZero ( )
```

Sets the camera 'roll' to be level with the XY plane

6.3.3.19 SetCameraToSprite()

```
void Blotch.BlGraphicsDeviceManager.SetCameraToSprite (
            BlSprite sprite )
```

Sets the camera position and orientation to the current position and orientation of a sprite. You could use for cockpit view, for example. Note that the camera will lag sprite movement unless the following is done: For every frame you must first calculate the sprite's position and orientation, call this function, and then draw everything.

Parameters

sprite	The sprite that the camera should be connected to

6.3.3.20 SetSpriteToCamera()

```
void Blotch.BlGraphicsDeviceManager.SetSpriteToCamera (
            BlSprite sprite )
```

Sets a sprite's BlSprite::Matrix to the current camera position and orientation. You could use this to implement a HUD, for example. Note: This only works correctly if the sprite has no parent (and is thus drawn directly) or it's parents are untransformed. If all you want is to set the sprite's position (but NOT orientation) to the camera, then set the sprite's Matrix.Translation = graphics.Eye

Parameters

sprite	The sprite that should be connected to the camera

6.3.3.21 TextToTexture()

```
Texture2D Blotch.BlGraphicsDeviceManager.TextToTexture (
            string text,
            SpriteFont font,
            Microsoft.Xna.Framework.Color?  color = null,
            Microsoft.Xna.Framework.Color?  backColor = null )
```

Returns a Texture2D containing the specified text. It's up to the caller to Dispose the returned texture.

Parameters

text	The text to write to the texture
font	Font to use
color	If specified, color of the text. (Default is white)
backColor	If specified, background color, like Color.Transparent. If null, then do not clear the background)

Returns

The texture (as a RenderTarget2D). Caller is responsible for Disposing this!

6.3.4 Member Data Documentation

6.3.4.1 AmbientLightColor

```
Vector3 Blotch.BlGraphicsDeviceManager.AmbientLightColor = new Vector3(.1f, .1f, .1f)
```

The ambient light color. If null, no ambient light is enabled. Note: There is no ambient color for a BlSprite. Both diffuse and ambient light illuminates the model's Color. See the BlSprite::Color member.

6.3.4.2 Aspect

```
double Blotch.BlGraphicsDeviceManager.Aspect =2
```

The aspect ratio.

6.3.4.3 AutoRotate

```
double Blotch.BlGraphicsDeviceManager.AutoRotate = 0
```

How fast DoDefaultGui should auto-rotate the scene.

6.3.4.4 CameraSpeed

```
double Blotch.BlGraphicsDeviceManager.CameraSpeed = .4
```

The responsiveness of the camera position to changes in TargetEye and TargetLookAt. A value of 0 means it doesn't respond to changes, 1 means it immediately responds. See Eye and LookAt for more information.

6.3.4.5 CameraUp

```
Vector3 Blotch.BlGraphicsDeviceManager.CameraUp
```

Camera Up vector. Initially set to +Z. ResetCamera and SetCameraToSprite updates this.

6.3.4.6 ClearColor

```
Microsoft.Xna.Framework.Color Blotch.BlGraphicsDeviceManager.ClearColor =new Microsoft.Xna.↩
Framework.Color(0,0,.1f)
```

The background color.

6.3.4.7 ClipRangeExcess

```
double Blotch.BlGraphicsDeviceManager.ClipRangeExcess = 5
```

Increase far clipping and decrease near clipping by this much. Use this to alleviate certain auto-clipping artifacts. See NearClip and FarClip.

6.3.4.8 DefGuiMaxLookZ

```
double Blotch.BlGraphicsDeviceManager.DefGuiMaxLookZ = 1
```

Caues DoDefaultGui to prevent the Z component of CameraForwardNormalized from rising above this value. For example, set this to zero so that DoDefaultGui won't allow the camera to look upward

6.3.4.9 DefGuiMinLookZ

```
double Blotch.BlGraphicsDeviceManager.DefGuiMinLookZ = -1
```

Causes DoDefaultGui to prevent the Z component of CameraForwardNormalized from falling below this value. For example, set this to zero so that DoDefaultGui won't allow the camera to look downward

6.3.4.10 DepthStencilStateDisabled

```
DepthStencilState Blotch.BlGraphicsDeviceManager.DepthStencilStateDisabled
```

Initial value:

```
= new DepthStencilState()
        {
            DepthBufferEnable = false,
            DepthBufferWriteEnable = false,
            DepthBufferFunction = CompareFunction.Always
        }
```

Assign DepthStencilState to this to disable depth buffering

6.3.4.11 DepthStencilStateEnabled

```
DepthStencilState Blotch.BlGraphicsDeviceManager.DepthStencilStateEnabled
```

Initial value:

```
= new DepthStencilState()
        {
            DepthBufferEnable = true,
            DepthBufferWriteEnable = true,
            DepthBufferFunction = CompareFunction.LessEqual
        }
```

Assign DepthStencilState to this to enable depth buffering

6.3.4.12 FarClip

```
double Blotch.BlGraphicsDeviceManager.FarClip = 0
```

The far clipping plane. If 0 then autoclip. If negative then auto-clip up to a limit of -FarClip. (Auto-clipping has a one-frame latency. A mechanism is employed in the camera control methods to somewhat alleviate this, but in certain cases you may still see a one frame dropout in visibility.) See the description for the depth buffer for more information.

6.3.4.13 FogColor

```
Vector3 Blotch.BlGraphicsDeviceManager.FogColor = null
```

If not null, color of fog.

6.3.4.14 fogEnd

```
float Blotch.BlGraphicsDeviceManager.fogEnd = 10
```

How far away fog ends. See FogColor.

6.3.4.15 fogStart

```
float Blotch.BlGraphicsDeviceManager.fogStart = 1
```

How far away fog starts. See FogColor.

6.3.4.16 FramePeriod

```
double Blotch.BlGraphicsDeviceManager.FramePeriod = 1/60.0
```

How much time between consecutive frames.

6.3.4.17 IsDisposed

```
bool Blotch.BlGraphicsDeviceManager.IsDisposed = false
```

Set when the object is Disposed.

6.3.4.18 Lights

```
List<Light> Blotch.BlGraphicsDeviceManager.Lights = new List<Light>()
```

The directional lights. Note: The BasicEffect shader only supports the first three. To handle more lights, you'll need to write your own shader.

6.3.4.19 NearClip

```
double Blotch.BlGraphicsDeviceManager.NearClip = 0
```

The near clipping plane. If 0 then autoclip. If negative then auto-clip down to a limit of -NearClip. (Auto-clipping has a one-frame latency. A mechanism is employed in the camera control methods to somewhat alleviate this, but in certain cases you may still see a one frame dropout in visibility.) See the description for the depth buffer for more information.

6.3.4.20 Projection

```
Microsoft.Xna.Framework.Matrix Blotch.BlGraphicsDeviceManager.Projection
```

The Projection matrix. Normally you would use the higher-level functions Zoom, Aspect, NearClip, or FarClip intead of changing this directly.

6.3.4.21 SpriteBatch

```
SpriteBatch Blotch.BlGraphicsDeviceManager.SpriteBatch =null
```

A SpriteBatch for use by certain text and texture drawing methods.

6.3.4.22 TargetEye

```
Vector3 Blotch.BlGraphicsDeviceManager.TargetEye
```

The point that Eye migrates to, according to CameraSpeed. This is normally controlled by DoDefaultGui, but can also be controlled by the AdjustCameraxxx methods. The easiest way to control the camera exactly (including camera roll), is to use SetCameraToSprite and then set the sprite's matrix as desired.

6.3.4.23 TargetLookAt

```
Vector3 Blotch.BlGraphicsDeviceManager.TargetLookAt
```

The point that LookAt migrates to, according to CameraSpeed. This is normally controlled by DoDefaultGui, but can also be controlled by the AdjustCameraxxx methods. The easiest way to control the camera exactly (including camera roll), is to use SetCameraToSprite and then set the sprite's matrix as desired.

6.3.4.24 View

```
Microsoft.Xna.Framework.Matrix Blotch.BlGraphicsDeviceManager.View
```

This is the view matrix. Normally you would use the higher-level functions Eye, LookAt, CameraUp, SetCameraToSprite, and DoDefaultGui intead of changing this directly.

6.3.4.25 Window

```
BlWindow3D Blotch.BlGraphicsDeviceManager.Window
```

The BlWindow3D associated with this object.

6.3.4.26 Zoom

```
double Blotch.BlGraphicsDeviceManager.Zoom =45
```

The field of view, in degrees.

6.3.5 Property Documentation

6.3.5.1 CameraForward

```
Vector3 Blotch.BlGraphicsDeviceManager.CameraForward [get]
```

The vector between Eye and LookAt. Writes to Eye and LookAt and calls to SetCameraToSprite cause this to be updated. Also see CameraForwardNormalized and CameraForwardMag.

6.3.5.2 CameraForwardMag

```
float Blotch.BlGraphicsDeviceManager.CameraForwardMag [get]
```

The magnitude of CameraForward. Writes to Eye and LookAt, and calls to SetCameraToSprite cause this to be updated. Also see CameraForward and CameraForwardNormalized.

6.3.5.3 CameraForwardNormalized

```
Vector3 Blotch.BlGraphicsDeviceManager.CameraForwardNormalized  [get]
```

Normalized form of CameraForward. Writes to Eye and LookAt, and calls to SetCameraToSprite cause this to be updated. Also see CameraForward and CameraForwardMag.

6.3.5.4 CameraRight

```
Vector3 Blotch.BlGraphicsDeviceManager.CameraRight  [get]
```

Camera Right vector. Writes to Eye and LookAt, and calls to SetCameraToSprite cause this to be updated.

6.3.5.5 CurrentAspect

```
double Blotch.BlGraphicsDeviceManager.CurrentAspect  [get]
```

Current aspect ratio. Same as Aspect unless Aspect==0.

6.3.5.6 CurrentFarClip

```
double Blotch.BlGraphicsDeviceManager.CurrentFarClip  [get]
```

Current value of far clipping plane. See FarClip.

6.3.5.7 CurrentNearClip

```
double Blotch.BlGraphicsDeviceManager.CurrentNearClip  [get]
```

Current value of near clipping plane. See NearClip.

6.3.5.8 Eye

```
Vector3 Blotch.BlGraphicsDeviceManager.Eye  [get]
```

The current camera position. See TargetEye.

6.3.5.9 LookAt

```
Vector3 Blotch.BlGraphicsDeviceManager.LookAt  [get]
```

The current camera LookAt position. See TargetLookAt.

6.3.5.10 MaxCamDistance

```
double Blotch.BlGraphicsDeviceManager.MaxCamDistance  [get]
```

Distance to the farthest sprite, plus its radius. Note this is set to a very small number by PrepareDraw, and then as BlWindow3D::FrameDraw is called it is set more reasonably.

6.3.5.11 MinCamDistance

```
double Blotch.BlGraphicsDeviceManager.MinCamDistance  [get]
```

Distance to the nearest sprite, less its radius. Note this is set to a very large number by PrepareDraw, and then as BlWindow3D::FrameDraw is called it is set more reasonably.

The documentation for this class was generated from the following file:

- C:/Users/kloum/Desktop/Source/Blotch3D/src/BlGraphicsDeviceManager.cs

6.4 Blotch.BlGuiControl Class Reference

A 2D GUI control. To create a GUI control: instantiate one of these, set its initial Texture, window position, and delegate, and then add it to BlWindow3D::GuiControls. (Any member can be dynamically changed.) The texture will be displayed, and then each frame the mouse is over it the delegate will be called. The delegate typically would examine the current mouse state (Mouse.GetState()) and the PrevMouseState member to detect button changes, etc. and perform an action. The delegate is called in the context of the window's 3D thread after the BlWindow3D::FrameProc method.

Public Member Functions

- delegate void OnMouseChangeDelegate (BlGuiControl guiCtrl)

  Delegates for a BlGuiControl are of this type
- **BlGuiControl** (BlWindow3D window)
- bool HandleInput ()

  Periodically called by BlWindow3D. You shouldn't need to call this.

Public Attributes

- Texture2D Texture = null

  The texture to display for this control. Don't forget to dispose it when done.
- Vector2 Position = Vector2.Zero

  The pixel position of this control in the BlWindow3D
- OnMouseChangeDelegate OnMouseOver = null

  The delegate to call each frame (from the 3D thread) when the mouse is over the control. A typical delegate would make a decision according to PrevMouseState and the current mouse state (Mouse.GetState).
- MouseState PrevMouseState = new MouseState()

  The previous mouse state. A delegate typically uses this along with the current mouse state to make a decision.
- BlWindow3D Window = null

  The window this BlGuiControl is in.

6.4.1 Detailed Description

A 2D GUI control. To create a GUI control: instantiate one of these, set its initial Texture, window position, and delegate, and then add it to BlWindow3D::GuiControls. (Any member can be dynamically changed.) The texture will be displayed, and then each frame the mouse is over it the delegate will be called. The delegate typically would examine the current mouse state (Mouse.GetState()) and the PrevMouseState member to detect button changes, etc. and perform an action. The delegate is called in the context of the window's 3D thread after the BlWindow3D::FrameProc method.

6.4.2 Member Function Documentation

6.4.2.1 HandleInput()

```
bool Blotch.BlGuiControl.HandleInput ( )
```

Periodically called by BlWindow3D. You shouldn't need to call this.

Returns

True if mouse is over any control, false otherwise.

6.4.2.2 OnMouseChangeDelegate()

```
delegate void Blotch.BlGuiControl.OnMouseChangeDelegate (
            BlGuiControl guiCtrl )
```

Delegates for a BlGuiControl are of this type

Parameters

guiCtrl	

6.4.3 Member Data Documentation

6.4.3.1 OnMouseOver

```
OnMouseChangeDelegate Blotch.BlGuiControl.OnMouseOver = null
```

The delegate to call each frame (from the 3D thread) when the mouse is over the control. A typical delegate would make a decision according to PrevMouseState and the current mouse state (Mouse.GetState).

6.4.3.2 Position

```
Vector2 Blotch.BlGuiControl.Position = Vector2.Zero
```

The pixel position of this control in the BlWindow3D

6.4.3.3 PrevMouseState

```
MouseState Blotch.BlGuiControl.PrevMouseState = new MouseState()
```

The previous mouse state. A delegate typically uses this along with the current mouse state to make a decision.

6.4.3.4 Texture

```
Texture2D Blotch.BlGuiControl.Texture = null
```

The texture to display for this control. Don't forget to dispose it when done.

6.4.3.5 Window

```
BlWindow3D Blotch.BlGuiControl.Window = null
```

The window this BlGuiControl is in.

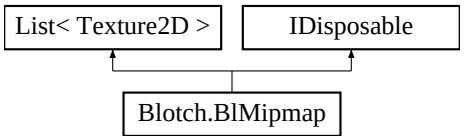
The documentation for this class was generated from the following file:

- C:/Users/kloum/Desktop/Source/Blotch3D/src/BlGuiControl.cs

6.5 Blotch.BlMipmap Class Reference

A BlMipmap holds a list of different resolutions of a texture, where one is applied to a sprite, depending on the ApparentSize of that sprite. You would assign it to a BlSprite::Mipmap. Note that this is a software mipmap (i.e. it isn't implemented in the 3D hardware). That is, unlike a hardware mipmap where different resolutions of the texture may appear on different parts of the sprite, only one resolution texture is used at time.

Inheritance diagram for Blotch.BlMipmap:



Public Member Functions

- BlMipmap (BlGraphicsDeviceManager graphics, Texture2D tex, int numMaps=999, bool reverseX=false, bool reverseY=false)

  Creates the mipmaps from the specified texture.
- void Dispose ()

  When finished with the object, you must call Dispose() from the same thread that created the object. You can call this multiple times, but once is enough.

Public Attributes

- bool IsDisposed = false

  Set when the object is Disposed.

6.5.1 Detailed Description

A BlMipmap holds a list of different resolutions of a texture, where one is applied to a sprite, depending on the ApparentSize of that sprite. You would assign it to a BlSprite::Mipmap. Note that this is a software mipmap (i.e. it isn't implemented in the 3D hardware). That is, unlike a hardware mipmap where different resolutions of the texture may appear on different parts of the sprite, only one resolution texture is used at time.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 BlMipmap()

```
Blotch.BlMipmap.BlMipmap (
            BlGraphicsDeviceManager graphics,
            Texture2D tex,
            int numMaps = 999,
            bool reverseX = false,
            bool reverseY = false )
```

Creates the mipmaps from the specified texture.

Parameters

graphics	Graphics device (typically the one owned by your BlWindow3D)
tex	Texture from which to create mipmaps, typically gotten from BlGraphics::LoadFromImageFile.
numMaps	Maximum number of mipmaps to create (none are created with lower resolution than 16x16)
reverseX	Whether to reverse pixels horizontally
reverseY	Whether to reverse pixels vertically

6.5.3 Member Function Documentation

6.5.3.1 Dispose()

```
void Blotch.BlMipmap.Dispose ( )
```

When finished with the object, you must call Dispose() from the same thread that created the object. You can call this multiple times, but once is enough.

6.5.4 Member Data Documentation

6.5.4.1 IsDisposed

```
bool Blotch.BlMipmap.IsDisposed = false
```

Set when the object is Disposed.

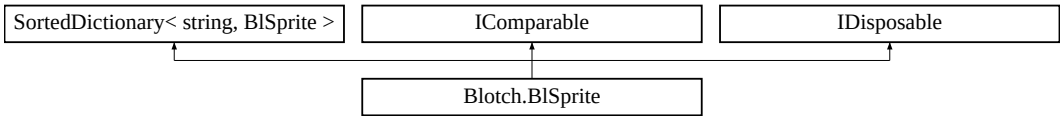
The documentation for this class was generated from the following file:

- C:/Users/kloum/Desktop/Source/Blotch3D/src/BlMipmap.cs

6.6 Blotch.BlSprite Class Reference

A BlSprite is a single 3D object. Each sprite can also hold any number of subsprites, so you can make a sprite tree (a scene graph). Subsprites are drawn in the order of their sorted names. Child sprites 'follow' the orientation and position of the parent sprite. That is, they exist in the coordinate system of the parent sprite. The location and orientation of a sprite in its parent's coordinate system is defined by the sprite's Matrix member. Subsprites, LODs, and Mipmap are NOT disposed when the sprite is disposed, so you can assign the same one to multiple sprites.

Inheritance diagram for Blotch.BlSprite:



Public Types

- enum PreDrawCmd { PreDrawCmd.Continue, PreDrawCmd.Abort, PreDrawCmd.UseCurrentAbsoluteMatrix }

  Return code from PreDraw callback. This tells Draw what to do next.
- enum PreSubspritesCmd { PreSubspritesCmd.Continue, PreSubspritesCmd.Abort, PreSubspritesCmd.DontDrawSubsprites }

  Return code from PreSubsprites callback. This tells Draw what to do next.
- enum SetEffectCmd { SetEffectCmd.Continue, SetEffectCmd.Abort, SetEffectCmd.Skip }

  Return code from PreSubsprites callback. This tells Draw what to do next.
- enum PreLocalCmd { PreLocalCmd.Continue, PreLocalCmd.Abort }

  Return code from PreSubsprites callback. This tells Draw what to do next.

Public Member Functions

- delegate void FrameProcType (BlSprite sprite)

  See #FrameProc
- void ExecuteFrameProc ()

  Execute the FrameProc, if it was specified in the BlSprite constructor. (Normally you wouldn't need to call this because its automatically called by the BlWindow.)
- delegate PreDrawCmd PreDrawType (BlSprite sprite)

  See PreDraw
- delegate PreSubspritesCmd PreSubspritesType (BlSprite sprite)

  See PreSubsprites
- delegate Effect SetMeshEffectType (BlSprite sprite, Effect effect)

  See SetEffect
- delegate PreLocalCmd PreLocalType (BlSprite sprite)

  See PreLocal
- delegate void DrawCleanupType (BlSprite sprite)

  See DrawCleanup
- BlSprite (BlGraphicsDeviceManager graphicsIn, string name, FrameProcType frameProc=null)

  Constructs a sprite
- void Add (BlSprite s)

  Add a subsprite. (A BlSprite inherits from a Dictionary of BlSprites. This wrapper method to the dictionary's Add method simply adds the sprite where the key is the sprite's Name.)
- Vector2 GetViewCoords ()

  Returns the current 2D view coordinates of the sprite (for passing to DrawText, for example), or null if it's behind the camera.
- void SetAllMaterialBlack ()

  Sets all material colors to black.
- double DoesRayIntersect (Ray ray)

  Returns the distance along the ray to the first point the ray enters the bounding sphere (BoundSphere), or null if it doesn't enter the sphere.
- List< BlSprite > GetRayIntersections (Ray ray, ulong flags=0xFFFFFFFFFFFFFFFF, List< BlSprite > sprites=null)

  Returns a list of subsprites that the ray hit (i.e. those that were within their radius of the ray)
- void Draw (Matrix? worldMatrixIn=null, ulong flagsIn=0xFFFFFFFFFFFFFFFF)

  Draws the sprite and the subsprites.
- Texture2D GetCurrentTexture ()

  If Mipmap is a BlMipmap, this returns the mimap texture that should currently be applied to the sprite. If Mipmap is a Texture2D, then that texture is returned.
- void SetupBasicEffect (BasicEffect effect)

Sets up in the specified BasicEffect all matrices and lighting parameters for this sprite. BlSprite::DrawInternal calls this for the BasicEffects embedded in the LOD models. For BlBasicEffect objects, see the overload of this method.

- void SetupBasicEffect (BlBasicEffect effect)

  Sets up in the specified BlBasicEffect with all matrices and lighting parameters for this sprite. App code might call this from a SetEffect delegate if, for example, it is using one of the BlBasicEffectxxx effects, like the BlBasicEffectWithAlphaTest.

- override string **ToString** ()
- int CompareTo (object obj)

  This makes a Sort operation sort sprites far to near. That is, the nearer sprites are later in the list. For sorting near to far, use something like myList.Sort(new Comparison<EsSprite>((b, a) => a.CompareTo(b)));

- void Dispose ()

  When finished with the object, you should call Dispose() from the same thread that created the object. You can call this multiple times, but once is enough. If it isn't called before the object becomes inaccessible, then the destructor will call it and, if BlDebug.EnableDisposeErrors is true (it is true by default for Debug builds), then it will get an exception saying that it wasn't called by the same thread that created it. This is because the platform's underlying 3D library (OpenGL, etc.) often requires 3D resources to be managed only by one thread.

Static Public Member Functions

- static Vector3 NearestPointOnLine (Vector3 point1, Vector3 point2, Vector3 nearPoint)

  Returns the point on the line between point1 and point2 that is nearest to nearPoint

Public Attributes

- ulong Flags = 0xFFFFFFFFFFFFFFFF

  The Flags field can be used by callbacks of Draw (PreDraw, PreSubsprites, PreLocal, and SetEffect) to indicate various user attributes of the sprite. Also, GetRayIntersections won't hit if the bitwise AND of this value and the flags argument passed to it is zero.

- double Alpha = 1

  Overall alpha of the current texture. (Not to be confused with pixel alpha)

- BlSprite Parent = null

  When you use the Add() method of a parent sprite to add a child sprite, the child sprite's Parent field is set to the Parent. If you add a child sprite in any other way, and you want the child's Parent field to reflect the parent, you'll have to assign it, yourself. This field is solely for reading by application code. It is not read by Blotch3D code.You might need this, for example, to implement particles via the FrameProc delegate. See particles example.

- List< object > LODs = new List<object>()

  The objects (levels of detail) to draw for this sprite. Only one element is drawn depending on the ApparentSize. Each element can be a Model, a VertexBuffer, or null (indicating nothing should be drawn for that LOD). Elements with lower indices are higher LODs. So index 0 has the highest detail, index 1 is second highest, etc. LOD decreases (the index increases) for every halving of the object's apparent size. You can adjust how close the LODs must be to the camera with LodScale. When the calculated LOD index is higher than the last element, then the last element is used. So the simplest way to use this is to add a single object to the list. You can add multiple references of the same object so multiple consecutive LODs draw the same object. You can set an element to null so it doesn't draw anything, which is typically the last element. A model can be assigned to multiple sprites. These are NOT disposed when the sprite is disposed.

- double LodScale = 9

  Defines the LOD scaling. The higher this value, the closer you must be to see a given LOD. A value of 9 (default) indicates that the highest LOD (LODs[0]) occurs when an object with a diameter of 1 roughly fills the window. Set to a large negative value, like -1000, to disable LODs (i.e. always use the highest resolution LOD).

- object Mipmap = null

A BIMipmap or a single Texture2D object. The model must include texture coordinates for this to be visible. It must also include normals if lighting other than 'emissive' is desired. If it's a BIMipmap, it will work the same as LODs (see LODs for more information). Specifically, the mipmap texture applied depends on the apparent size of the model. The next higher mipmap is used for every doubling of model size, where element zero is the highest resolution, used when the apparent size is largest. If a mipmap is not available for the apparent size, the next higher available on is used. Most graphics subsystems do support mipmaps, but these are software mipmaps, so only one image is used over a model for a given model apparent size rather than nearer portions of the model showing higher-level mipmaps. This is NOT disposed when the sprite is disposed, so a given BIMipmap or Texture2D may be assigned to multiple sprites.

- double MipmapScale = 5

  Defines the mipmap (Textures) scaling. The higher this value, the closer you must be to see a given mipmap. Set to a large negative value, like -1000, to disable mipmaps (i.e. always use the highest resolution mipmap).

- BoundingSphere BoundSphere = null

  The bounding sphere for this sprite. This is automatically updated when a model is drawn, but not if vertices are drawn. In that case you should set/update it explicitly if any of the internal functions may need it to be roughly correct, like if auto-clipping is enabled or a mouse selection or ray may hit the sprite and the hit be properly detected.

- bool SphericalBillboard = false

  Spherically billboard the model. Specifically, keep the model's 'forward' direction pointing at the camera and keep its 'Up' direction pointing in the same direction as the camera's 'Up' direction. Also see CylindricalBillboardX, CylindricalBillboardY, CylindricalBillboardZ, and ConstSize.

- Vector3 CylindricalBillboardX = Vector3.Zero

  If non-zero, this is the rotation vector and magnitude of cylindrical billboarding where the angle calculation assumes this vector is the X axis, even though it may not be. The more this varies from that axis, the more eccentric the billboarding behavior. The amount of billboarding is equal to: 2∗mag^2 - 1/mag^2. So if this vector's magnitude is unity (1), then full cylindrical billboarding occurs. A vector magnitude of 0.605 produces double reverse cylindrical billboarding. Also see SphericalBillboard, CylindricalBillboardY, CylindricalBillboardZ, and ConstSize.

- Vector3 CylindricalBillboardY = Vector3.Zero

  If non-zero, this is the rotation vector and magnitude of cylindrical billboarding where the angle calculation assumes this vector is the Y axis, even though it may not be. The more this varies from that axis, the more eccentric the billboarding behavior. The amount of billboarding is equal to: 2∗mag^2 - 1/mag^2. So if this vector's magnitude is unity (1), then full cylindrical billboarding occurs. A vector magnitude of 0.605 produces double reverse cylindrical billboarding. Also see SphericalBillboard, CylindricalBillboardX, CylindricalBillboardZ, and ConstSize.

- Vector3 CylindricalBillboardZ = Vector3.Zero

  If non-zero, this is the rotation vector and magnitude of cylindrical billboarding where the angle calculation assumes this vector is the Z axis, even though it may not be. The more this varies from that axis, the more eccentric the billboarding behavior. The amount of billboarding is equal to: 2∗mag^2 - 1/mag^2. So if this vector's magnitude is unity (1), then full cylindrical billboarding occurs. A vector magnitude of 0.605 produces double reverse cylindrical billboarding. Also see SphericalBillboard, CylindricalBillboardX, CylindricalBillboardY, and ConstSize.

- bool ConstSize = false

  If true, maintain a constant apparent size for the sprite regardless of camera distance or zoom. This is typically used along with one of the Billboarding effects (see SphericalBillboard, CylindricalBillboardX, etc.). Note that if ConstSize is true, ApparentSize, LodScale, and MipmapScale still act as if it is false, and therefore in that case you may want to disable them (set them to large negative values). If both ConstSize and any Billboarding is enabled and you have asymmetric scaling (different scaling for each dimension), then you'll need to separate those operations into different levels of the sprite tree to obtain the desired behavior. You'll also probably want to disable the depth stencil buffer and control which sprite is drawn first so that certain sprites are 'always on top'. See the examples.

- Matrix AbsoluteMatrix = Matrix.Identity

  The Draw method takes an incoming 'world' matrix parameter which is the coordinate system of its parent. AbsoluteMatrix is that incoming world matrix parameter times the Matrix member and altered according to Billboarding and ConstSize. This is not read-only because a callback (see PreDraw, PreSubsprites, PreLocal, and SetEffect) may need to change it from within the Draw method. This is the matrix that is also passed to subsprites as their 'world' matrix.

- Matrix Matrix = Matrix.Identity

  The matrix for this sprite. This defines the sprite's orientation and position relative to the parent coordinate system. For more detailed information, see AbsoluteMatrix.

- BIGraphicsDeviceManager Graphics = null

  Current incoming graphics parameter to the Draw method. Typically this would be of interest to a callback function (see PreDraw, PreSubsprites, PreLocal, and SetEffect).

- Matrix LastWorldMatrix = null

Current incoming world matrix parameter to the Draw method. Typically this would be of interest to a callback function (see PreDraw, PreSubsprites, PreLocal, and SetEffect).

- bool IncludeInAutoClipping = true

  Wwhether to participate in autoclipping calculations, when they are enabled.

- ulong FlagsParameter = 0

  Current incoming flags parameter to the Draw method. Typically this would be of interest to a callback function (see PreDraw, PreSubsprites, PreLocal, and SetEffect).

- Vector3 Color = new Vector3(.5f, 1, .5f)

  The color of the material. This is lit by both diffuse and ambient light. If null, MonoGame's default color is kept.

- Vector3 EmissiveColor = new Vector3(.1f, .1f, .1f)

  The emissive color. If null, MonoGame's default is kept.

- Vector3 SpecularColor = null

  The specular color. If null, MonoGame's default is kept.

- float SpecularPower = 4

  If a specular color is specified, this is the specular power.

- PreDrawType PreDraw = null

  If not null, Draw method calls this at the beginning before doing anything else. From this function one might examine and/or alter any public writable BlSprite field, and/or control the further execution of the Draw method.

- PreSubspritesType PreSubsprites = null

  If not null, Draw method calls this after the matrix calculations for AbsoluteMatrix (including billboards, CamDistance, ConstSize, etc.) but before drawing the subsprites or local model. From this function one might examine and/or alter any public writable BlSprite field.

- SetMeshEffectType SetEffect = null

  If this not null, then the Draw method executes this delegate for each model mesh effect instead using the default BasicEffects. See the SpriteAlphaTexture for an example. If you use this but the effect is still of type BlBasicEffect, then you might want to call SetupBasicEffect from within this delegate to set all the effect's parameters, rather than doing it yourself. The return value is the new or altered effect. If this is called when the thing to draw is a Vertex↩ PositionNormalTexture, then the effect parameter passed in is a null.

- PreLocalType PreLocal = null

  If not null, Draw method calls this after drawing subsprites (if appropriate) but before drawing the local model. From this function one might examine and/or alter any public writable BlSprite field, and/or abort the Draw method.

- DrawCleanupType DrawCleanup = null

  If not null, Draw method calls this at the end.

- string Name

  The name of the BlSprite

- bool IsDisposed = false

  Set when the object is Disposed.

Properties

- double ApparentSize `[get]`

  This is proportional to the apparent 2D size of the sprite. (Calculated from the last Draw operation that occurred, but before any effect of ConstSize)

- double LodTarget `[get]`

  This read-only value is the log of the reciprocal of ApparentSize. It is used in the calculation of the LOD and the mipmap level. See LODs and Mipmap for more information.

- double CamDistance `[get]`

  Distance to the camera.

- double PrevCamDistance `[get]`

  Internal use only. Used for predicting new depth clipping values

6.6.1 Detailed Description

A BlSprite is a single 3D object. Each sprite can also hold any number of subsprites, so you can make a sprite tree (a scene graph). Subsprites are drawn in the order of their sorted names. Child sprites 'follow' the orientation and position of the parent sprite. That is, they exist in the coordinate system of the parent sprite. The location and orientation of a sprite in its parent's coordinate system is defined by the sprite's Matrix member. Subsprites, LODs, and Mipmap are NOT disposed when the sprite is disposed, so you can assign the same one to multiple sprites.

6.6.2 Member Enumeration Documentation

6.6.2.1 PreDrawCmd

`enum Blotch.BlSprite.PreDrawCmd [strong]`

Return code from PreDraw callback. This tells Draw what to do next.

Enumerator

Continue	Continue Draw method execution
Abort	Draw should immediately return
UseCurrentAbsoluteMatrix	Continue Draw method execution, but don't bother re-calculating AbsoluteMatrix. One would typically return this if, for example, its known that AbsoluteMatrix will not change from its current value because the Draw parameters will be the same as they were the last time Draw was called. This happens, for example, when multiple calls are being made in the same draw iteration for graphic operations that require multiple passes.

6.6.2.2 PreLocalCmd

`enum Blotch.BlSprite.PreLocalCmd [strong]`

Return code from PreSubsprites callback. This tells Draw what to do next.

Enumerator

Continue	Continue Draw method execution
Abort	Draw should immediately return

6.6.2.3 PreSubspritesCmd

`enum Blotch.BlSprite.PreSubspritesCmd [strong]`

Return code from PreSubsprites callback. This tells Draw what to do next.

Enumerator

Continue	Continue Draw method execution
Abort	Draw should immediately return
DontDrawSubsprites	Skip drawing subsprites

6.6.2.4 SetEffectCmd

```
enum Blotch.BlSprite.SetEffectCmd  [strong]
```

Return code from PreSubsprites callback. This tells Draw what to do next.

Enumerator

Continue	Continue Draw method execution for the mesh
Abort	Draw should immediately return
Skip	Draw should skip the current mesh

6.6.3 Constructor & Destructor Documentation

6.6.3.1 BlSprite()

```
Blotch.BlSprite.BlSprite (
            BlGraphicsDeviceManager graphicsIn,
            string name,
            FrameProcType frameProc = null )
```

Constructs a sprite

Parameters

graphics↩ In	The BlGraphicsDeviceManager that operates on this sprite
name	The name of the sprite (must be unique among other sprites in the same subsprite list)
frameProc	The delegate to run every frame. (Note that this is NOT called from within the Draw method. Rather, it is called separately only once every frame. Therefore you can, for example, delete and add other subsprites of this sprite's Parent. See Parent for more information.)

6.6.4 Member Function Documentation

6.6.4.1 Add()

```
void Blotch.BlSprite.Add (
            BlSprite s )
```

Add a subsprite. (A BlSprite inherits from a Dictionary of BlSprites. This wrapper method to the dictionary's Add method simply adds the sprite where the key is the sprite's Name.)

Parameters

s	

6.6.4.2 CompareTo()

```
int Blotch.BlSprite.CompareTo (
            object obj )
```

This makes a Sort operation sort sprites far to near. That is, the nearer sprites are later in the list. For sorting near to far, use something like myList.Sort(new Comparison<EsSprite>((b, a) => a.CompareTo(b)));

Parameters

obj	

Returns

6.6.4.3 Dispose()

```
void Blotch.BlSprite.Dispose ( )
```

When finished with the object, you should call Dispose() from the same thread that created the object. You can call this multiple times, but once is enough. If it isn't called before the object becomes inaccessible, then the destructor will call it and, if BlDebug.EnableDisposeErrors is true (it is true by default for Debug builds), then it will get an exception saying that it wasn't called by the same thread that created it. This is because the platform's underlying 3D library (OpenGL, etc.) often requires 3D resources to be managed only by one thread.

6.6.4.4 DoesRayIntersect()

```
double Blotch.BlSprite.DoesRayIntersect (
            Ray ray )
```

Returns the distance along the ray to the first point the ray enters the bounding sphere (BoundSphere), or null if it doesn't enter the sphere.

Parameters

ray	

Returns

How far along the ray till the first intersection, or null oif it didn't intersect

6.6.4.5 Draw()

```
void Blotch.BlSprite.Draw (
            Matrix?  worldMatrixIn = null,
            ulong flagsIn = 0xFFFFFFFFFFFFFFFF )
```

Draws the sprite and the subsprites.

Parameters

worldMatrixIn	Defines the position and orientation of the sprite
flagsIn	Copied to LastFlags for use by any callback of Draw (PreDraw, PreSubspriteDraw, PreLocalDraw, and SetEffect) that wants it

6.6.4.6 DrawCleanupType()

```
delegate void Blotch.BlSprite.DrawCleanupType (
            BlSprite sprite )
```

See DrawCleanup

Parameters

sprite	

6.6.4.7 ExecuteFrameProc()

```
void Blotch.BlSprite.ExecuteFrameProc ( )
```

Execute the FrameProc, if it was specified in the BlSprite constructor. (Normally you wouldn't need to call this because its automatically called by the BlWindow.)

6.6.4.8 FrameProcType()

```
delegate void Blotch.BlSprite.FrameProcType (
            BlSprite sprite )
```

See #FrameProc

Parameters

sprite	

6.6.4.9 GetCurrentTexture()

```
Texture2D Blotch.BlSprite.GetCurrentTexture ( )
```

If Mipmap is a BlMipmap, this returns the mimap texture that should currently be applied to the sprite. If Mipmap is a Texture2D, then that texture is returned.

Returns

6.6.4.10 GetRayIntersections()

```
List<BlSprite> Blotch.BlSprite.GetRayIntersections (
            Ray ray,
            ulong flags = 0xFFFFFFFFFFFFFFFF,
            List< BlSprite > sprites = null )
```

Returns a list of subsprites that the ray hit (i.e. those that were within their radius of the ray)

Parameters

ray	The ray we are searching
flags	Check for a hit only if flags & BlSprite::Flags is non-zero
sprites	An existing sprite list to load. If null, then this allocates a new sprite list.

Returns

A list of subsprites that the ray hit

6.6.4.11 GetViewCoords()

```
Vector2 Blotch.BlSprite.GetViewCoords ( )
```

Returns the current 2D view coordinates of the sprite (for passing to DrawText, for example), or null if it's behind the camera.

Returns

The view coords of the sprite

6.6.4.12 NearestPointOnLine()

```
static Vector3 Blotch.BlSprite.NearestPointOnLine (
            Vector3 point1,
            Vector3 point2,
            Vector3 nearPoint )  [static]
```

Returns the point on the line between point1 and point2 that is nearest to nearPoint

Parameters

point1	
point2	
nearPoint	

Returns

Point on the line nearest to nearPoint

6.6.4.13 PreDrawType()

```
delegate PreDrawCmd Blotch.BlSprite.PreDrawType (
            BlSprite sprite )
```

See PreDraw

Parameters

sprite	

Returns

6.6.4.14 PreLocalType()

```
delegate PreLocalCmd Blotch.BlSprite.PreLocalType (
            BlSprite sprite )
```

See PreLocal

Parameters

sprite	

Returns

6.6.4.15 PreSubspritesType()

```
delegate PreSubspritesCmd Blotch.BlSprite.PreSubspritesType (
            BlSprite sprite )
```

See PreSubsprites

Parameters

sprite	

Returns

6.6.4.16 SetAllMaterialBlack()

```
void Blotch.BlSprite.SetAllMaterialBlack ( )
```

Sets all material colors to black.

6.6.4.17 SetMeshEffectType()

```
delegate Effect Blotch.BlSprite.SetMeshEffectType (
            BlSprite sprite,
            Effect effect )
```

See SetEffect

Parameters

sprite	
effect	

Returns

6.6.4.18 SetupBasicEffect() [1/2]

```
void Blotch.BlSprite.SetupBasicEffect (
            BasicEffect effect )
```

Sets up in the specified BasicEffect all matrices and lighting parameters for this sprite. BlSprite::DrawInternal calls this for the BasicEffects embedded in the LOD models. For BlBasicEffect objects, see the overload of this method.

6.6.4.19 SetupBasicEffect() [2/2]

```
void Blotch.BlSprite.SetupBasicEffect (
            BlBasicEffect effect )
```

Sets up in the specified BlBasicEffect with all matrices and lighting parameters for this sprite. App code might call this from a SetEffect delegate if, for example, it is using one of the BlBasicEffectxxx effects, like the BlBasicEffect↩ WithAlphaTest.

6.6.5 Member Data Documentation

6.6.5.1 AbsoluteMatrix

```
Matrix Blotch.BlSprite.AbsoluteMatrix = Matrix.Identity
```

The Draw method takes an incoming 'world' matrix parameter which is the coordinate system of its parent. AbsoluteMatrix is that incoming world matrix parameter times the Matrix member and altered according to Billboarding and ConstSize. This is not read-only because a callback (see PreDraw, PreSubsprites, PreLocal, and SetEffect) may need to change it from within the Draw method. This is the matrix that is also passed to subsprites as their 'world' matrix.

6.6.5.2 Alpha

```
double Blotch.BlSprite.Alpha = 1
```

Overall alpha of the current texture. (Not to be confused with pixel alpha)

6.6.5.3 BoundSphere

```
BoundingSphere Blotch.BlSprite.BoundSphere = null
```

The bounding sphere for this sprite. This is automatically updated when a model is drawn, but not if vertices are drawn. In that case you should set/update it explicitly if any of the internal functions may need it to be roughly correct, like if auto-clipping is enabled or a mouse selection or ray may hit the sprite and the hit be properly detected.

6.6.5.4 Color

```
Vector3 Blotch.BlSprite.Color = new Vector3(.5f, 1, .5f)
```

The color of the material. This is lit by both diffuse and ambient light. If null, MonoGame's default color is kept.

6.6.5.5 ConstSize

```
bool Blotch.BlSprite.ConstSize = false
```

If true, maintain a constant apparent size for the sprite regardless of camera distance or zoom. This is typically used along with one of the Billboarding effects (see SphericalBillboard, CylindricalBillboardX, etc.). Note that if ConstSize is true, ApparentSize, LodScale, and MipmapScale still act as if it is false, and therefore in that case you may want to disable them (set them to large negative values). If both ConstSize and any Billboarding is enabled and you have asymmetric scaling (different scaling for each dimension), then you'll need to separate those operations into different levels of the sprite tree to obtain the desired behavior. You'll also probably want to disable the depth stencil buffer and control which sprite is drawn first so that certain sprites are 'always on top'. See the examples.

6.6.5.6 CylindricalBillboardX

```
Vector3 Blotch.BlSprite.CylindricalBillboardX = Vector3.Zero
```

If non-zero, this is the rotation vector and magnitude of cylindrical billboarding where the angle calculation assumes this vector is the X axis, even though it may not be. The more this varies from that axis, the more eccentric the billboarding behavior. The amount of billboarding is equal to: 2∗mag^2 - 1/mag^2. So if this vector's magnitude is unity (1), then full cylindrical billboarding occurs. A vector magnitude of 0.605 produces double reverse cylindrical billboarding. Also see SphericalBillboard, CylindricalBillboardY, CylindricalBillboardZ, and ConstSize.

6.6.5.7 CylindricalBillboardY

```
Vector3 Blotch.BlSprite.CylindricalBillboardY = Vector3.Zero
```

If non-zero, this is the rotation vector and magnitude of cylindrical billboarding where the angle calculation assumes this vector is the Y axis, even though it may not be. The more this varies from that axis, the more eccentric the billboarding behavior. The amount of billboarding is equal to: 2∗mag^2 - 1/mag^2. So if this vector's magnitude is unity (1), then full cylindrical billboarding occurs. A vector magnitude of 0.605 produces double reverse cylindrical billboarding. Also see SphericalBillboard, CylindricalBillboardX, CylindricalBillboardZ, and ConstSize.

6.6.5.8 CylindricalBillboardZ

```
Vector3 Blotch.BlSprite.CylindricalBillboardZ = Vector3.Zero
```

If non-zero, this is the rotation vector and magnitude of cylindrical billboarding where the angle calculation assumes this vector is the Z axis, even though it may not be. The more this varies from that axis, the more eccentric the billboarding behavior. The amount of billboarding is equal to: 2∗mag^2 - 1/mag^2. So if this vector's magnitude is unity (1), then full cylindrical billboarding occurs. A vector magnitude of 0.605 produces double reverse cylindrical billboarding. Also see SphericalBillboard, CylindricalBillboardX, CylindricalBillboardY, and ConstSize.

6.6.5.9 DrawCleanup

```
DrawCleanupType Blotch.BlSprite.DrawCleanup = null
```

If not null, Draw method calls this at the end.

6.6.5.10 EmissiveColor

```
Vector3 Blotch.BlSprite.EmissiveColor = new Vector3(.1f, .1f, .1f)
```

The emissive color. If null, MonoGame's default is kept.

6.6.5.11 Flags

```
ulong Blotch.BlSprite.Flags = 0xFFFFFFFFFFFFFFFF
```

The Flags field can be used by callbacks of Draw (PreDraw, PreSubsprites, PreLocal, and SetEffect) to indicate various user attributes of the sprite. Also, GetRayIntersections won't hit if the bitwise AND of this value and the flags argument passed to it is zero.

6.6.5.12 FlagsParameter

```
ulong Blotch.BlSprite.FlagsParameter = 0
```

Current incoming flags parameter to the Draw method. Typically this would be of interest to a callback function (see PreDraw, PreSubsprites, PreLocal, and SetEffect).

6.6.5.13 Graphics

```
BlGraphicsDeviceManager Blotch.BlSprite.Graphics = null
```

Current incoming graphics parameter to the Draw method. Typically this would be of interest to a callback function (see PreDraw, PreSubsprites, PreLocal, and SetEffect).

6.6.5.14 IncludeInAutoClipping

```
bool Blotch.BlSprite.IncludeInAutoClipping = true
```

Wwhether to participate in autoclipping calculations, when they are enabled.

6.6.5.15 IsDisposed

```
bool Blotch.BlSprite.IsDisposed = false
```

Set when the object is Disposed.

6.6.5.16 LastWorldMatrix

```
Matrix Blotch.BlSprite.LastWorldMatrix = null
```

Current incoming world matrix parameter to the Draw method. Typically this would be of interest to a callback function (see PreDraw, PreSubsprites, PreLocal, and SetEffect).

6.6.5.17 LODs

```
List<object> Blotch.BlSprite.LODs = new List<object>()
```

The objects (levels of detail) to draw for this sprite. Only one element is drawn depending on the ApparentSize. Each element can be a Model, a VertexBuffer, or null (indicating nothing should be drawn for that LOD). Elements with lower indices are higher LODs. So index 0 has the highest detail, index 1 is second highest, etc. LOD decreases (the index increases) for every halving of the object's apparent size. You can adjust how close the LODs must be to the camera with LodScale. When the calculated LOD index is higher than the last element, then the last element is used. So the simplest way to use this is to add a single object to the list. You can add multiple references of the same object so multiple consecutive LODs draw the same object. You can set an element to null so it doesn't draw anything, which is typically the last element. A model can be assigned to multiple sprites. These are NOT disposed when the sprite is disposed.

6.6.5.18 LodScale

```
double Blotch.BlSprite.LodScale = 9
```

Defines the LOD scaling. The higher this value, the closer you must be to see a given LOD. A value of 9 (default) indicates that the highest LOD (LODs[0]) occurs when an object with a diameter of 1 roughly fills the window. Set to a large negative value, like -1000, to disable LODs (i.e. always use the highest resolution LOD).

6.6.5.19 Matrix

```
Matrix Blotch.BlSprite.Matrix = Matrix.Identity
```

The matrix for this sprite. This defines the sprite's orientation and position relative to the parent coordinate system. For more detailed information, see AbsoluteMatrix.

6.6.5.20 Mipmap

```
object Blotch.BlSprite.Mipmap = null
```

A BlMipmap or a single Texture2D object. The model must include texture coordinates for this to be visible. It must also include normals if lighting other than 'emissive' is desired. If it's a BlMipmap, it will work the same as LODs (see LODs for more information). Specifically, the mipmap texture applied depends on the apparent size of the model. The next higher mipmap is used for every doubling of model size, where element zero is the highest resolution, used when the apparent size is largest. If a mipmap is not available for the apparent size, the next higher available on is used. Most graphics subsystems do support mipmaps, but these are software mipmaps, so only one image is used over a model for a given model apparent size rather than nearer portions of the model showing higher-level mipmaps. This is NOT disposed when the sprite is disposed, so a given BlMipmap or Texture2D may be assigned to multiple sprites.

6.6.5.21 MipmapScale

```
double Blotch.BlSprite.MipmapScale = 5
```

Defines the mipmap (Textures) scaling. The higher this value, the closer you must be to see a given mipmap. Set to a large negative value, like -1000, to disable mipmaps (i.e. always use the highest resolution mipmap).

6.6.5.22 Name

```
string Blotch.BlSprite.Name
```

The name of the BlSprite

6.6.5.23 Parent

```
BlSprite Blotch.BlSprite.Parent = null
```

When you use the Add() method of a parent sprite to add a child sprite, the child sprite's Parent field is set to the Parent. If you add a child sprite in any other way, and you want the child's Parent field to reflect the parent, you'll have to assign it, yourself. This field is solely for reading by application code. It is not read by Blotch3D code.You might need this, for example, to implement particles via the FrameProc delegate. See particles example.

6.6.5.24 PreDraw

```
PreDrawType Blotch.BlSprite.PreDraw = null
```

If not null, Draw method calls this at the beginning before doing anything else. From this function one might examine and/or alter any public writable BlSprite field, and/or control the further execution of the Draw method.

6.6.5.25 PreLocal

```
PreLocalType Blotch.BlSprite.PreLocal = null
```

If not null, Draw method calls this after drawing subsprites (if appropriate) but before drawing the local model. From this function one might examine and/or alter any public writable BlSprite field, and/or abort the Draw method.

6.6.5.26 PreSubsprites

```
PreSubspritesType Blotch.BlSprite.PreSubsprites = null
```

If not null, Draw method calls this after the matrix calculations for AbsoluteMatrix (including billboards, CamDistance, ConstSize, etc.) but before drawing the subsprites or local model. From this function one might examine and/or alter any public writable BlSprite field.

6.6.5.27 SetEffect

```
SetMeshEffectType Blotch.BlSprite.SetEffect = null
```

If this not null, then the Draw method executes this delegate for each model mesh effect instead using the default BasicEffects. See the SpriteAlphaTexture for an example. If you use this but the effect is still of type BlBasicEffect, then you might want to call SetupBasicEffect from within this delegate to set all the effect's parameters, rather than doing it yourself. The return value is the new or altered effect. If this is called when the thing to draw is a VertexPositionNormalTexture, then the effect parameter passed in is a null.

6.6.5.28 SpecularColor

```
Vector3 Blotch.BlSprite.SpecularColor = null
```

The specular color. If null, MonoGame's default is kept.

6.6.5.29 SpecularPower

```
float Blotch.BlSprite.SpecularPower = 4
```

If a specular color is specified, this is the specular power.

6.6.5.30 SphericalBillboard

```
bool Blotch.BlSprite.SphericalBillboard = false
```

Spherically billboard the model. Specifically, keep the model's 'forward' direction pointing at the camera and keep its 'Up' direction pointing in the same direction as the camera's 'Up' direction. Also see CylindricalBillboardX, CylindricalBillboardY, CylindricalBillboardZ, and ConstSize.

6.6.6 Property Documentation

6.6.6.1 ApparentSize

```
double Blotch.BlSprite.ApparentSize [get]
```

This is proportional to the apparent 2D size of the sprite. (Calculated from the last Draw operation that occurred, but before any effect of ConstSize)

6.6.6.2 CamDistance

```
double Blotch.BlSprite.CamDistance [get]
```

Distance to the camera.

6.6.6.3 LodTarget

```
double Blotch.BlSprite.LodTarget [get]
```

This read-only value is the log of the reciprocal of ApparentSize. It is used in the calculation of the LOD and the mipmap level. See LODs and Mipmap for more information.

6.6.6.4 PrevCamDistance

```
double Blotch.BlSprite.PrevCamDistance [get]
```

Internal use only. Used for predicting new depth clipping values

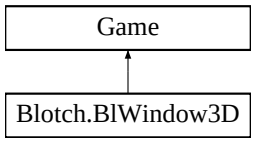
The documentation for this class was generated from the following file:

- C:/Users/kloum/Desktop/Source/Blotch3D/src/BlSprite.cs

6.7 Blotch.BlWindow3D Class Reference

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 open 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, and not return until the window closes. All code that accesses 3D resources must be done in that thread, including code that creates and uses all Blotch3D and MonoGame objects. Note that this rule also applies to any code structure that may internally use other threads, as well. Do not use Parallel, async, etc. code structures that access 3D resources. Other threads that need to access 3D resources can do so by passing a delegate to EnqueueCommand and EnqueueCommandBlocking.

Inheritance diagram for Blotch.BlWindow3D:



Public Member Functions

- delegate void Command (BlWindow3D win)

  See EnqueueCommand, EnqueueCommandBlocking, and BlWindow3D for more info
- BlWindow3D ()

  See BlWindow3D for details.
- void EnqueueCommand (Command cmd)

  Since all operations accessing 3D resources must be done by the 3D thread, this allows other threads to send commands to execute in the 3D thread. For example, you might need another thread to be able to create, move, and delete BlSprites. You can also use this for general thread safety of various operations. This method does not block. Also see BlWindow3D and the (blocking) EnqueueCommandBlocking for more details.
- void EnqueueCommandBlocking (Command cmd)

  Since all operations accessing 3D resources must be done by the 3D thread, this allows other threads to send commands to execute in the 3D thread. For example, you might need another thread to be able to create, move, and delete BlSprites. You can also use this for general thread safety of various operations. This method blocks until the command has executed. Also see BlWindow3D and the (non-blocking) EnqueueCommand for more details.
- void FrameProcSpritesAdd (BlSprite s)

  Used internally
- void FrameProcSpritesRemove (BlSprite s)

  Used internally
- new void Dispose ()

  When finished with the object, you should call Dispose() from the same thread that created the object. You can call this multiple times, but once is enough. If it isn't called before the object becomes inaccessible, then the destructor will call it and, if BlDebug.EnableDisposeErrors is true (it is true by default for Debug builds), then it will get an exception saying that it wasn't called by the same thread that created it. This is because the platform's underlying 3D library (OpenGL, etc.) often requires 3D resources to be managed only by one thread.

Public Attributes

- BlGraphicsDeviceManager Graphics

  The BlGraphicsDeviceManager associated with this window. This is automatically created when you create the BlWindow3D.
- ConcurrentDictionary< string, BlGuiControl > GuiControls = new ConcurrentDictionary<string, BlGuiControl>()

  The GUI controls for this window. See BlGuiControl for details.
- bool IsDisposed = false

  Set when the object is Disposed.

Protected Member Functions

- override void Initialize ()

  Used internally, Do NOT override. Use Setup instead.
- override void LoadContent ()

  Used internally, Do NOT override. Use Setup instead.
- virtual void Setup ()

  Override this and put all initialization and global content creation code in it. See BlWindow3D for details.
- override void Update (GameTime timeInfo)

  Used internally, Do NOT override. Use FrameProc instead.
- virtual void FrameProc (GameTime timeInfo)

  See BlWindow3D for details.
- override void Draw (GameTime timeInfo)

  Used internally, Do NOT override. Use FrameDraw instead.
- virtual void FrameDraw (GameTime timeInfo)

  See BlWindow3D for details.

6.7.1 Detailed Description

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 open 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, and not return until the window closes. All code that accesses 3D resources must be done in that thread, including code that creates and uses all Blotch3D and MonoGame objects. Note that this rule also applies to any code structure that may internally use other threads, as well. Do not use Parallel, async, etc. code structures that access 3D resources. Other threads that need to access 3D resources can do so by passing a delegate to EnqueueCommand and EnqueueCommandBlocking.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 BlWindow3D()

```
Blotch.BlWindow3D.BlWindow3D ( )
```

See BlWindow3D for details.

6.7.3 Member Function Documentation

6.7.3.1 Command()

```
delegate void Blotch.BlWindow3D.Command (
            BlWindow3D win )
```

See EnqueueCommand, EnqueueCommandBlocking, and BlWindow3D for more info

Parameters

win	The BlWindow3D object

6.7.3.2 Dispose()

```
new void Blotch.BlWindow3D.Dispose ( )
```

When finished with the object, you should call Dispose() from the same thread that created the object. You can call this multiple times, but once is enough. If it isn't called before the object becomes inaccessible, then the destructor will call it and, if BlDebug.EnableDisposeErrors is true (it is true by default for Debug builds), then it will get an exception saying that it wasn't called by the same thread that created it. This is because the platform's underlying 3D library (OpenGL, etc.) often requires 3D resources to be managed only by one thread.

6.7.3.3 Draw()

```
override void Blotch.BlWindow3D.Draw (
            GameTime timeInfo )  [protected]
```

Used internally, Do NOT override. Use FrameDraw instead.

Parameters

timeInfo	

6.7.3.4 EnqueueCommand()

```
void Blotch.BlWindow3D.EnqueueCommand (
            Command cmd )
```

Since all operations accessing 3D resources must be done by the 3D thread, this allows other threads to send commands to execute in the 3D thread. For example, you might need another thread to be able to create, move, and delete BlSprites. You can also use this for general thread safety of various operations. This method does not block. Also see BlWindow3D and the (blocking) EnqueueCommandBlocking for more details.

Parameters

cmd	

6.7.3.5 EnqueueCommandBlocking()

```
void Blotch.BlWindow3D.EnqueueCommandBlocking (
            Command cmd )
```

Since all operations accessing 3D resources must be done by the 3D thread, this allows other threads to send commands to execute in the 3D thread. For example, you might need another thread to be able to create, move, and delete BlSprites. You can also use this for general thread safety of various operations. This method blocks until the command has executed. Also see BlWindow3D and the (non-blocking) EnqueueCommand for more details.

Parameters

cmd	

6.7.3.6 FrameDraw()

```
virtual void Blotch.BlWindow3D.FrameDraw (
            GameTime timeInfo )  [protected], [virtual]
```

See BlWindow3D for details.

Parameters

timeInfo	

6.7.3.7 FrameProc()

```
virtual void Blotch.BlWindow3D.FrameProc (
            GameTime timeInfo )  [protected], [virtual]
```

See BlWindow3D for details.

Parameters

timeInfo	

6.7.3.8 FrameProcSpritesAdd()

```
void Blotch.BlWindow3D.FrameProcSpritesAdd (
            BlSprite s )
```

Used internally

Parameters

s	

6.7.3.9 FrameProcSpritesRemove()

```
void Blotch.BlWindow3D.FrameProcSpritesRemove (
            BlSprite s )
```

Used internally

Parameters

s	

6.7.3.10 Initialize()

```
override void Blotch.BlWindow3D.Initialize ( )  [protected]
```

Used internally, Do NOT override. Use Setup instead.

6.7.3.11 LoadContent()

```
override void Blotch.BlWindow3D.LoadContent ( )  [protected]
```

Used internally, Do NOT override. Use Setup instead.

6.7.3.12 Setup()

```
virtual void Blotch.BlWindow3D.Setup ( )  [protected], [virtual]
```

Override this and put all initialization and global content creation code in it. See BlWindow3D for details.

6.7.3.13 Update()

```
override void Blotch.BlWindow3D.Update (
            GameTime timeInfo )  [protected]
```

Used internally, Do NOT override. Use FrameProc instead.

Parameters

timeInfo	

6.7.4 Member Data Documentation

6.7.4.1 Graphics

```
BlGraphicsDeviceManager Blotch.BlWindow3D.Graphics
```

The BlGraphicsDeviceManager associated with this window. This is automatically created when you create the BlWindow3D.

6.7.4.2 GuiControls

```
ConcurrentDictionary<string, BlGuiControl> Blotch.BlWindow3D.GuiControls = new Concurrent↩
Dictionary<string, BlGuiControl>()
```

The GUI controls for this window. See BlGuiControl for details.

6.7.4.3 IsDisposed

```
bool Blotch.BlWindow3D.IsDisposed = false
```

Set when the object is Disposed.

The documentation for this class was generated from the following file:

- C:/Users/kloum/Desktop/Source/Blotch3D/src/BlWindow3D.cs

6.8 Blotch.BlGraphicsDeviceManager.Light Class Reference

Defines a light. See the Lights field. The default BasicShader supports up to three lights.

Public Attributes

- Vector3 **LightDirection** = new Vector3(1, 0, 0)
- Vector3 **LightDiffuseColor** = new Vector3(1, 0, 1)
- Vector3 **LightSpecularColor** = new Vector3(0, 1, 0)

6.8.1 Detailed Description

Defines a light. See the Lights field. The default BasicShader supports up to three lights.

The documentation for this class was generated from the following file:

- C:/Users/kloum/Desktop/Source/Blotch3D/src/BlGraphicsDeviceManager.cs