

Blotch3D Manual

Contents

1	Blotch3D	1
1.1	Quick start	1
1.2	Introduction	1
1.3	Project structure	3
1.4	Development	4
1.5	Making 3D models	5
1.6	Dynamically changing a sprite's orientation and position	6
1.7	Matrix internals	6
1.8	A Short Glossary of 3D Graphics Terms	9
1.9	Troubleshooting	11
1.10	Rights	12
2	Namespace Index	13
2.1	Packages	13
3	Hierarchical Index	15
3.1	Class Hierarchy	15
4	Class Index	17
4.1	Class List	17
5	Namespace Documentation	19
5.1	Blotch Namespace Reference	19

6	Class Documentation	21
6.1	Blotch.BIGraphicsDeviceManager Class Reference	21
6.1.1	Detailed Description	24
6.1.2	Constructor & Destructor Documentation	24
6.1.2.1	BIGraphicsDeviceManager()	24
6.1.3	Member Function Documentation	25
6.1.3.1	AdjustCameraDolly()	25
6.1.3.2	AdjustCameraPan()	25
6.1.3.3	AdjustCameraRotation()	25
6.1.3.4	AdjustCameraTruck()	26
6.1.3.5	AdjustCameraZoom()	26
6.1.3.6	CalculateRay()	26
6.1.3.7	CloneTexture2D()	27
6.1.3.8	Dispose()	27
6.1.3.9	DoDefaultGui()	27
6.1.3.10	DrawText()	28
6.1.3.11	DrawTexture()	29
6.1.3.12	ExtendClippingTo()	29
6.1.3.13	GetWindowCoordinates()	29
6.1.3.14	Initialize()	30
6.1.3.15	LoadFromImageFile()	30
6.1.3.16	PrepareDraw()	30
6.1.3.17	ResetCamera()	31
6.1.3.18	SetCameraRollToZero()	31
6.1.3.19	SetCameraToSprite()	31
6.1.3.20	SetSpriteToCamera()	31
6.1.3.21	TextToTexture()	32
6.1.4	Member Data Documentation	32
6.1.4.1	AmbientLightColor	32
6.1.4.2	Aspect	32

6.1.4.3	AutoRotate	33
6.1.4.4	CameraSpeed	33
6.1.4.5	CameraUp	33
6.1.4.6	ClearColor	33
6.1.4.7	DefGuiMaxLookZ	33
6.1.4.8	DefGuiMinLookZ	33
6.1.4.9	DepthStencilStateDisabled	34
6.1.4.10	DepthStencilStateEnabled	34
6.1.4.11	FarClip	34
6.1.4.12	FogColor	34
6.1.4.13	fogEnd	35
6.1.4.14	fogStart	35
6.1.4.15	FramePeriod	35
6.1.4.16	IsDisposed	35
6.1.4.17	Lights	35
6.1.4.18	NearClip	35
6.1.4.19	Projection	36
6.1.4.20	SpriteBatch	36
6.1.4.21	TargetEye	36
6.1.4.22	TargetLookAt	36
6.1.4.23	View	36
6.1.4.24	Window	36
6.1.4.25	Zoom	37
6.1.5	Property Documentation	37
6.1.5.1	CameraForward	37
6.1.5.2	CameraForwardMag	37
6.1.5.3	CameraForwardNormalized	37
6.1.5.4	CameraRight	37
6.1.5.5	CurrentAspect	38
6.1.5.6	CurrentFarClip	38

6.1.5.7	CurrentNearClip	38
6.1.5.8	Eye	38
6.1.5.9	LookAt	38
6.1.5.10	MaxCamDistance	38
6.1.5.11	MinCamDistance	39
6.2	Blotch.BIGuiControl Class Reference	39
6.2.1	Detailed Description	40
6.2.2	Member Function Documentation	40
6.2.2.1	HandleInput()	40
6.2.2.2	OnMouseChangeDelegate()	40
6.2.3	Member Data Documentation	40
6.2.3.1	OnMouseOver	40
6.2.3.2	Position	41
6.2.3.3	PrevMouseState	41
6.2.3.4	Texture	41
6.2.3.5	Window	41
6.3	Blotch.BIMipmap Class Reference	41
6.3.1	Detailed Description	42
6.3.2	Constructor & Destructor Documentation	42
6.3.2.1	BIMipmap()	42
6.3.3	Member Function Documentation	43
6.3.3.1	Dispose()	43
6.3.4	Member Data Documentation	43
6.3.4.1	IsDisposed	43
6.4	Blotch.BISprite Class Reference	43
6.4.1	Detailed Description	47
6.4.2	Member Enumeration Documentation	47
6.4.2.1	PreDrawCmd	47
6.4.2.2	PreLocalCmd	48
6.4.2.3	PreSubspritesCmd	48

6.4.2.4	SetMeshEffectCmd	48
6.4.3	Constructor & Destructor Documentation	49
6.4.3.1	BISprite()	49
6.4.4	Member Function Documentation	49
6.4.4.1	Add()	49
6.4.4.2	CompareTo()	50
6.4.4.3	Dispose()	50
6.4.4.4	DoesRayIntersect()	50
6.4.4.5	Draw()	51
6.4.4.6	DrawCleanupType()	51
6.4.4.7	ExecuteFrameProc()	51
6.4.4.8	FrameProcType()	51
6.4.4.9	GetRayIntersections()	52
6.4.4.10	GetViewCoords()	52
6.4.4.11	NearestPointOnLine()	52
6.4.4.12	PreDrawType()	53
6.4.4.13	PreLocalType()	53
6.4.4.14	PreSubspritesType()	53
6.4.4.15	SetAllMaterialBlack()	54
6.4.4.16	SetMeshEffectType()	54
6.4.5	Member Data Documentation	54
6.4.5.1	AbsoluteMatrix	54
6.4.5.2	BoundSphere	55
6.4.5.3	Color	55
6.4.5.4	ConstSize	55
6.4.5.5	CylindricalBillboardX	55
6.4.5.6	CylindricalBillboardY	55
6.4.5.7	CylindricalBillboardZ	56
6.4.5.8	DrawCleanup	56
6.4.5.9	EmissiveColor	56

6.4.5.10	Flags	56
6.4.5.11	FlagsParameter	56
6.4.5.12	Graphics	57
6.4.5.13	IncludeInAutoClipping	57
6.4.5.14	IsDisposed	57
6.4.5.15	LastWorldMatrix	57
6.4.5.16	LODs	57
6.4.5.17	LodScale	58
6.4.5.18	Matrix	58
6.4.5.19	Mipmap	58
6.4.5.20	MipmapScale	58
6.4.5.21	Name	58
6.4.5.22	PreDraw	59
6.4.5.23	PreLocal	59
6.4.5.24	PreSubsprites	59
6.4.5.25	SetMeshEffect	59
6.4.5.26	SpecularColor	59
6.4.5.27	SpecularPower	60
6.4.5.28	SphericalBillboard	60
6.4.6	Property Documentation	60
6.4.6.1	ApparentSize	60
6.4.6.2	CamDistance	60
6.4.6.3	LodTarget	60
6.5	Blotch.BIWindow3D Class Reference	61
6.5.1	Detailed Description	62
6.5.2	Constructor & Destructor Documentation	62
6.5.2.1	BIWindow3D()	62
6.5.3	Member Function Documentation	62
6.5.3.1	Command()	62
6.5.3.2	Dispose()	63

6.5.3.3	Draw()	63
6.5.3.4	EnqueueCommand()	63
6.5.3.5	EnqueueCommandBlocking()	64
6.5.3.6	FrameDraw()	64
6.5.3.7	FrameProc()	64
6.5.3.8	FrameProcSpritesAdd()	64
6.5.3.9	FrameProcSpritesRemove()	65
6.5.3.10	Initialize()	65
6.5.3.11	LoadContent()	65
6.5.3.12	Setup()	65
6.5.3.13	Update()	65
6.5.4	Member Data Documentation	66
6.5.4.1	Graphics	66
6.5.4.2	GuiControls	66
6.5.4.3	IsDisposed	66
6.6	Blotch.BIGraphicsDeviceManager.Light Class Reference	66
6.6.1	Detailed Description	66

Chapter 1

Blotch3D

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

1.1 Quick start

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

1.2 Introduction

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

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

- Load standard 3D model file types as "sprites", and display and move thousands of them in 3D at high frame rates.
- Set a sprite's material, texture, and lighting response.
- Load textures from standard image files.
- Show 2D and in-world (as a texture) text in any font, size, color, etc. at any 2D or 3D position, and make text follow a sprite in 2D or 3D. 2D textures and text can have a transparent background.
- Attach sprites to other sprites to create 'sprite trees' as large as you want. Child sprite orientation, position, scale, etc. are relative to the parent sprite, and can be changed dynamically (i.e. the sprite trees are dynamic scene graphs.)
- Override all steps in the drawing of each sprite.

- You can give the user easy control over all aspects of the camera (zoom, pan, truck, dolly, rotate, etc.).
- Easily control all aspects of the camera programmatically.
- Create billboard sprites.
- Connect sprites to the camera to implement HUD models and text.
- Connect the camera to a sprite to implement 'cockpit view', etc.
- Implement GUI controls (as dynamic 2D text or image rectangles) in the 3D window.
- Implement a skybox sprite.
- Get a list of sprites touching a ray, to implement weapons fire, etc.
- Get a list of sprites under the mouse position, to implement mouse selection, tooltips, pop-up menus, etc.
- Implement levels-of-detail.
- Implement mipmaps.
- Apply translucent textures to sprites (i.e. textures with an alpha channel) with certain restrictions (see [Sprite↔AlphaTexture](#) for details).
- Create sprite models programmatically (custom vertices).
- Use with WPF and WinForms, on Microsoft Windows.
- Access and override many window features and functions using the provided WinForms Form object of the window (Microsoft Windows only).
- Detect collisions between sprites.
- Implement fog
- Define ambient lighting, and up to three point-light sources. (More lights can be defined if a custom shader is used.)
- Build for many platforms (currently supports all Microsoft Windows platforms, iOS, Android, MacOS, Linux, PS4, PSVita, Xbox One, and Switch).

Blotch3D sits on top of MonoGame. MonoGame is a widely used 3D library for C#. It is free, fast, cross platform, actively developed by a large community, and used in many professional games. There is a plethora of MonoGame documentation, tutorials, examples, and discussions on line.

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

See [MonoGame.net](http://monogame.net) for the official MonoGame documentation. When searching on-line for other MonoGame documentation and discussions, be sure to note the MonoGame version being discussed. Documentation of earlier versions may not be compatible with the latest.

MonoGame fully implements Microsoft's (no longer supported) XNA 4 engine, but for multiple platforms. It also implements features beyond XNA 4. Therefore XNA 4 documentation you come across may not show you the best way to do something, and documentation of earlier versions of XNA (versions 2 and 3) will often not be correct. For conversion of XNA 3 to XNA 4 see <http://www.nelsonhurst.com/xna-3-1-to-xna-4-0-cheatsheet/>.

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

1.3 Project structure

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

"BlotchExample01\Basic" is a bare-bones Blotch3D application, where Example.cs contains the example code. Other example projects also contain an Example.cs, which is similar to the one from the basic example but with a few additions to it to demonstrate a certain feature. In fact, you can do a diff between the basic Examples.cs file and another example's source file to see what extra code must be added to implement the features it demonstrates [TBD: the "full" example needs to be split to several simpler examples].

All the provided projects are configured to build for the Microsoft Windows x64 platform. See below for other platforms.

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

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

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

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

Or you can:

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

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

1.4 Development

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

To make a 3D window, you must derive a class from `BIWindow3D` and override the `Setup`, `FrameProc`, and `FrameDraw` methods.

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

We will call the abovementioned thread the "3D thread".

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

This pattern and these rules are also used by `MonoGame`. In fact, the `BIWindow3D` class inherits from `MonoGame`'s "Game" class. But instead of overriding certain "Game" class methods, you override `BIWindow3D`'s `Setup`, `FrameProc`, and `FrameDraw` methods. Other "Game" class methods and events can still be overridden, if needed.

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

The `Setup` method is called by the 3D thread exactly once at the beginning of instantiation of the `BIWindow3D`-derived object. You might put time-consuming initialization of persistent things in there like the loading and initialization of persistent content (sprite models, fonts, `BISprites`, etc.).

The `FrameProc` method is called by the 3D thread once every frame. For single-threaded applications this is typically where the bulk of application code resides, except the actual drawing code. For multi-threaded applications, this is typically where all application code resides that does anything with 3D resources. (Note: You can also pass a delegate to the `BISprite` constructor, which will cause that delegate to be executed every time the `BIWindow3D`'s `FrameProc` method is executed. The effect is the same as putting the code in `FrameProc`, but it better encapsulates sprite-specific code.)

The `FrameDraw` method is called by the 3D thread every frame, but only if there is enough CPU for that thread. Otherwise it is called less frequently. This is where you must put drawing code (`BISprite.Draw`, `BIGraphicsDeviceManager.DrawText`, etc.). For apps that may suffer from severe CPU exhaustion (at least for the 3D thread), you may want to put your app code in this method so it is called less frequently (as long as that code can properly handle being called at variable rates).

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

If you are developing a multithreaded app, then you would probably want to reserve the 3D thread only for tasks that access 3D resources. When other threads do need to create, change, or destroy 3D resources or otherwise do something in a thread-safe way with the 3D thread, they can pass a delegate to `BIWindow3D.EnqueueCommand` or `BIWindow3D.EnqueueCommandBlocking`.

Because multiple windows are not conducive to some of the supported platforms, `MonoGame`, and thus `Blotch3D`, do not support more than one window. You *can* create multiple windows, but they don't work correctly (input sometimes goes to the wrong window) and in certain situations will crash. If you want to be able to "close" and "re-open" a window, you can just hide and show the same window. (On Microsoft Windows, you can use the `BIWindow3D.Form` object for that.)

Officially, `MonoGame` must create the 3D window, and does not allow you to specify an existing window to use as the 3D window. There are some platform-specific ways to do it described online, but note that they may not work in later `MonoGame` releases. To properly make the `MonoGame` window be a child window of an existing GUI, you need to explicitly size, position, and convey Z order to the original `MonoGame` window so that it is overlaid over the child window.

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

- The models you specify for a sprite object (see the `BlSprite.LODs` field) are MonoGame "Model" objects. So, you can, for example, specify custom shaders, etc., for those models.
- The `BlWindow3D` class derives from the MonoGame "Game" class.
- The `BlGraphicsDeviceManager` class derives from MonoGame's "GraphicsDeviceManager" class.

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

See the examples, reference documentation, and IntelliSense for more information.

1.5 Making 3D models

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

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

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

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

1. Open your application .csproj in an Editor.
2. In the first `<PropertyGroup>` section add `<MonoGamePlatform>$(Platform)</MonoGamePlatform>`, where `$(Platform)` is the system you are targeting e.g Windows, iOS, Android. For example: `<MonoGamePlatform>Windows</MonoGamePlatform>`
3. Add the following lines right underneath the `<MonoGamePlatform />` element: `<MonoGameInstallDirectory Condition="'$(OS)' != 'Unix' " >$(MSBuildProgramFiles32)</MonoGameInstallDirectory>`
`<MonoGameInstallDirectory Condition="'$(OS)' == 'Unix' " >$(MSBuildExtensionsPath)</MonoGameInstallDirectory>`
4. Find the `<Import/>` element for the CSharp (or FSharp) targets and underneath add:
`<Import Project="$(MSBuildExtensionsPath)\MonoGame\v3.0\MonoGame.Content.Builder.targets" />`

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

If no existing model meets your needs, you can either programmatically create a model by specifying the vertices and normals (see the example that uses custom Vertices), or create a model with, for example, the Blender 3D modeler and then add it to the project with the pipeline manager. The pipeline manager can import several model file types. You can also instruct Blender to include texture (UV) mapping by using one of the countless tutorials online, like <https://www.youtube.com/watch?v=2xTzJIaKQFY> or https://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/UV_Map_Basics. Also, you may be able to import certain existing models from the web, but mind the copyright.

1.6 Dynamically changing a sprite's orientation and position

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

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

There are also static and instance Matrix methods and operator overloads to combine (multiply) matrices to form a single matrix which combines the effects of multiple matrices. For example, a rotate matrix and a scale matrix can be multiplied to form a single rotate-scale matrix. But mind the multiplication order because matrix multiplication is not commutative. See below for details, but novices can simply try the operation one way (like A times B) and, if it doesn't work the way you wanted, do it the other way (B times A).

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

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

1.7 Matrix internals

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

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

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

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

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

For example,

$X' = 2X$ (where X is the initial value, and X' is the final value)

... which, when applied to each vertex, would change the above vertices from (4,1) and (3,3) to (8,1) and (6,3).

We might want to define how to change each X according to the original X value of each vertex *and also according to the original Y value*, like this:

$$X' = aX + bY$$

For example, if a=0 and b=1, then this would set the new X of each vertex to its original Y value.

Finally, we might also want to define how to create a new Y for each vertex according to its original X and original Y. So, the equations for both the new X and new Y are:

$$X' = aX + bY$$

$$Y' = cX + dY$$

(Remember, the idea is to apply this to every vertex.)

By convention we might write the four matrix elements (a, b, c, and d) in a 2x2 matrix, like this:

a b

c d

This should all be very easy to understand.

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

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

1 0

0 1

...then the vertices are unchanged, because...

$$X' = 1X + 0Y$$

$$Y' = 0X + 1Y$$

...sets X' to X and Y' to Y.

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

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

3 0

0 3

To scale only X by 3 (stretch a model in the X direction about the origin), then use the matrix:

3 0

0 1

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

1 0

0 -1

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

0 -1

1 0

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

0.707 -0.707

0.707 0. 707

Note that '0.707' is the sine of 45 degrees.

A matrix can be created to rotate any amount about any axis.

(The Matrix class provides functions that make it easy to create a rotation matrix from a rotation axis and angle, or pitch and yaw and roll, or something called a quaternion, since otherwise we'd have to call sine and cosine functions, ourselves, to create the matrix elements.)

Since we often also want to translate (move) points *without* regard to their current distances from the origin as we did at the beginning of this section, we add more numbers to the matrix just for that purpose. And since many mathematical operations on matrices work only if the matrix has the same number of rows as columns, we add more elements simply to make the rows and columns the same size. And since Blotch3D/MonoGame works in 3-space, we add even more numbers to handle the Z dimension. So, the final matrix size in 3D graphics is 4x4.

Specifically:

$$X' = aX + bY + cZ + d$$

$$Y' = eX + fY + gZ + h$$

$$Z' = iX + jY + kZ + l$$

$$W = mX + nY + oZ + p$$

(Consider the W as unused, for now.)

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

Rather than using the above 16 letters ('a' through 'p') for the matrix elements, the Matrix class in MonoGame uses the following field names:

M11 M12 M13 M14

M21 M22 M23 M24

M31 M32 M33 M34

M41 M42 M43 M44

Besides the ability to multiply entire matrices (as mentioned at the beginning of this section), you can also divide (i.e. multiply by a matrix inverse) matrices to, for example, solve for a matrix that was used in a previous matrix multiply, or otherwise isolate one operation from another. Welcome to linear algebra! The Matrix class provides matrix multiply, inversion, etc. methods. If you are interested in how the individual matrix elements are processed to perform matrix arithmetic, please look it up online.

As was previously mentioned, each sprite has a matrix describing how that sprite and its children are transformed from the parent sprite's coordinate system. Specifically, Blotch3D does a matrix-multiply of the parent's matrix with the child's matrix to create the final ("absolute") matrix used to draw that child, and that matrix is also used as the parent matrix for the subsprites of that child.

1.8 A Short Glossary of 3D Graphics Terms

Vertex

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

Polygon

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

Ambient lighting

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

Diffuse lighting

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

Texture

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

Normal

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

X-axis

The axis that extends right from the origin.

Y-axis

The axis that extends forward from the origin.

Z-axis

The axis that extends up from the origin.

Origin

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

Translation

Movement. The placing of something at a different location from its original location.

Rotation

The circular movement of each vertex of a model about the same axis.

Scale

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

Shear (skew)

A pulling of one side of a model in one direction, and the opposite side in the opposite direction, without rotation, such that the model is distorted rather than rotated. A parallelogram is a rectangle that has experienced shear. If you apply another shear along an orthogonal axis of the first shear, you rotate the model.

Yaw

Rotation about the Y-axis

Pitch

Rotation about the X-axis, after any Yaw has been applied.

Roll

Rotation about the Z-axis, after any Pitch has been applied.

Euler angles

The yaw, pitch, and roll of a model, applied in that order.

Matrix

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

Frame

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

Depth buffer

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

Near clipping plane (NearClip)

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

Far clipping plane (FarClip)

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

Model space

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

World space

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

View space

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

1.9 Troubleshooting

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

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

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

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

Q: I set a sprite's matrix so that one of the dimensions has a scale of zero, but then the sprite becomes black.

A: A sprite's matrix also affects its normals. By setting a dimension's scale to zero, you may have caused some of the normals to be zero'd-out as well.

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

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

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

A: The floating-point precision limitation of the depth buffer can cause this. Try increasing your near clip and/or decreasing your far clip so the depth buffer doesn't have to cover so much dynamic range.

Q: I have a sprite that I want always to be visible, but I think its invisible because its outside the depth buffer, but I don't want to change the depth buffer.

A: Try doing a `Graphics.GraphicsDevice.DepthStencilState = Graphics.DepthStencilState.Disabled` in the `BlSprite.PreDraw` delegate, and set it back to `DepthStencilState.Enabled` in the `BlSprite.DrawCleanup` delegate.

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

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

1.10 Rights

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

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

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

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

Chapter 2

Namespace Index

2.1 Packages

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

Blotch	19
----------------------------------	----

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

Blotch.BIGuiControl	39
Dictionary	
Blotch.BISprite	43
Game	
Blotch.BIWindow3D	61
GraphicsDeviceManager	
Blotch.BIGraphicsDeviceManager	21
ICloneable	
Blotch.BIGraphicsDeviceManager	21
IComparable	
Blotch.BISprite	43
IDisposable	
Blotch.BIMipmap	41
Blotch.BISprite	43
Blotch.BIGraphicsDeviceManager.Light	66
List	
Blotch.BIMipmap	41

Chapter 4

Class Index

4.1 Class List

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

Blotch.BIGraphicsDeviceManager	
This holds everything having to do with an output device. BIWindow3D creates one of these for itself.	21
Blotch.BIGuiControl	
A 2D GUI control. To create a GUI control: instantiate one of these, set its initial Texture (remember to create it in the 3D thread context), window position, and delegate, and then add it to BIWindow3D::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 BIWindow3D::FrameProc method. You can use BIGraphicsDeviceManager::TextToTexture to create a textual textures, or just load a texture from a content file. Remember to Dispose textures when you are done with them.	39
Blotch.BIMipmap	
A mipmap of textures for a given BISprite . You could load this from an image file and then assign it to a BISprite::Mipmap . Note that this is a software mipmap (i.e. it isn't implemented in the 3D hardware). That is, only one resolution texture is used at time.	41
Blotch.BISprite	
A BISprite is a single 3D object. Each sprite can also hold any number of subsprites, so you can make a sprite tree (a scene graph). In that case the 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.	43
Blotch.BIWindow3D	
To make a 3D window, you must derive a class from BIWindow3D 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	61
Blotch.BIGraphicsDeviceManager.Light	
Defines a light. See the Lights field. The default BasicShader supports up to three lights. . . .	66

Chapter 5

Namespace Documentation

5.1 Blotch Namespace Reference

Classes

- class **BIDebug**

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 it exist. For this reason you should first test your app as a debug build console app.

- class **BIGraphicsDeviceManager**

This holds everything having to do with an output device. [BIWindow3D](#) creates one of these for itself.

- class **BIGuiControl**

A 2D GUI control. To create a GUI control: instantiate one of these, set its initial Texture (remember to create it in the 3D thread context), window position, and delegate, and then add it to [BIWindow3D::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 [BIWindow3D::FrameProc](#) method. You can use [BIGraphicsDeviceManager::TextToTexture](#) to create a textual textures, or just load a texture from a content file. Remember to Dispose textures when you are done with them.

- class **BIMipmap**

A mipmap of textures for a given [BISprite](#). You could load this from an image file and then assign it to a [BISprite::Mipmap](#). Note that this is a software mipmap (i.e. it isn't implemented in the 3D hardware). That is, only one resolution texture is used at time.

- class **BISprite**

A [BISprite](#) is a single 3D object. Each sprite can also hold any number of subsprites, so you can make a sprite tree (a scene graph). In that case the 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 **BIWindow3D**

To make a 3D window, you must derive a class from [BIWindow3D](#) 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](#).

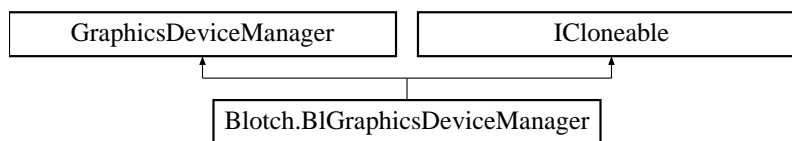
Chapter 6

Class Documentation

6.1 Blotch.BIGraphicsDeviceManager Class Reference

This holds everything having to do with an output device. [BIWindow3D](#) creates one of these for itself.

Inheritance diagram for Blotch.BIGraphicsDeviceManager:



Classes

- class [Light](#)
Defines a light. See the [Lights](#) field. The default BasicShader supports up to three lights.

Public Member Functions

- [BIGraphicsDeviceManager](#) ([BIWindow3D](#) window)
- void [Initialize](#) ()
For internal use only. Apps should not normally call this. This initializes some values AFTER the [BIWindow3D](#) has been created.
- void [ExtendClippingTo](#) ([BISprite](#) s)
Informs the auto-clipping code of an object that should be included in the clipping region. This is mainly for internal use. Application code should control clipping with [NearClip](#) and [FarClip](#).
- void [SetSpriteToCamera](#) ([BISprite](#) sprite)
Sets a sprite's [BISprite::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](#) ([BISprite](#) 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.
- 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](#) (BISprite 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)

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

This is automatically called once at the beginning of your [BIWindow3D::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 [BIWindow3D::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 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 [BIDebug::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

- Microsoft.Xna.Framework.Matrix [View](#)
This is the view matrix. Normally you would use the higher-level functions [Eye](#), [LookAt](#), [CameraUp](#), [SetCameraToSprite](#), and [DoDefaultGui](#) instead 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](#) instead 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](#). See [Eye](#) for more information.
- Vector3 [TargetLookAt](#)
The point that [LookAt](#) migrates to, according to [CameraSpeed](#). See [LookAt](#) for more information.
- 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, or 0 = autoclip.
- double [FarClip](#) = 0
The far clipping plane, or 0 = autoclip.
- 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](#).*
- [BIWindow3D Window](#)
The [BIWindow3D](#) associated with this object.
- [SpriteBatch](#) [SpriteBatch](#) = null
A [SpriteBatch](#) for use by certain text and teture 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. Note: To change the camera position, set [TargetEye](#). Also see [CameraSpeed](#).
- [Vector3](#) [LookAt](#) [get]
The current camera LookAt position. Note: To change the camera LookAt, set [TargetLookAt](#). Also see [CameraSpeed](#).
- [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 [BIWindow3D::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 [BIWindow3D::FrameDraw](#) is called it is set more reasonably.

6.1.1 Detailed Description

This holds everything having to do with an output device. [BIWindow3D](#) creates one of these for itself.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 BIGraphicsDeviceManager()

```
Blotch.BIGraphicsDeviceManager.BIGraphicsDeviceManager (
    BIWindow3D window )
```

Parameters

<i>window</i>	The BIWindow3D object for which this is to be the BIGraphicsDeviceManager
---------------	---

6.1.3 Member Function Documentation

6.1.3.1 AdjustCameraDolly()

```
void Blotch.BIGraphicsDeviceManager.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

<i>dif</i>	How much to dolly camera (plus = toward LookAt , minus = away)
------------	--

6.1.3.2 AdjustCameraPan()

```
void Blotch.BIGraphicsDeviceManager.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

<i>difX</i>	How much to pan horizontally
<i>difY</i>	How much to pan vertically

6.1.3.3 AdjustCameraRotation()

```
void Blotch.BIGraphicsDeviceManager.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

<i>difX</i>	How much to rotate the camera horizontally
<i>difY</i>	How much to rotate the camera vertically

6.1.3.4 AdjustCameraTruck()

```
void Blotch.BIGraphicsDeviceManager.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

<i>difX</i>	How much to truck the camera horizontally
<i>difY</i>	How much to truck the camera vertically

6.1.3.5 AdjustCameraZoom()

```
void Blotch.BIGraphicsDeviceManager.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

<i>dif</i>	How much to zoom camera (plus = magnify, minus = reduce)
------------	--

6.1.3.6 CalculateRay()

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

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

Parameters

<i>windowPosition</i>	The window's pixel coordinates
-----------------------	--------------------------------

Returns

The Ray into the window at the specified pixel coordinates

6.1.3.7 CloneTexture2D()

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

Returns a deepcopy of the texture

Parameters

<i>tex</i>	The texture to deepcopy
------------	-------------------------

Returns

A deepcopy of tex

6.1.3.8 Dispose()

```
new void Blotch.BIGraphicsDeviceManager.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 `BIDebug::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.

6.1.3.9 DoDefaultGui()

```
Ray Blotch.BIGraphicsDeviceManager.DoDefaultGui ( )
```

Updates [Eye](#), [LookAt](#), etc. according to mouse and certain key input. Specifically: Wheel=Dolly, CTRL+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.

Returns

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

6.1.3.10 DrawText()

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

Draws text on the window.

Parameters

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

6.1.3.11 DrawTexture()

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

Draws a texture in the window.

Parameters

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

6.1.3.12 ExtendClippingTo()

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

Informs the auto-clipping code of an object that should be included in the clipping region. This is mainly for internal use. Application code should control clipping with [NearClip](#) and [FarClip](#).

Parameters

<i>s</i>	The sprite that should be included in the auto-clipping code
----------	--

6.1.3.13 GetWindowCoordinates()

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

Returns the window coordinates of the specified sprite.

Parameters

<i>sprite</i>	The sprite to get the window coordinates of
---------------	---

Returns

The window coordinates of the sprite, in pixels

6.1.3.14 Initialize()

```
void Blotch.BIGraphicsDeviceManager.Initialize ( )
```

For internal use only. Apps should not normally call this. This initializes some values AFTER the [BIWindow3D](#) has been created.

6.1.3.15 LoadFromImageFile()

```
Texture2D Blotch.BIGraphicsDeviceManager.LoadFromImageFile (
    string fileName )
```

Loads a texture directly from an image file.

Parameters

<i>fileName</i>	An image file of any standard type supported by MonoGame (jpg, png, etc.)
-----------------	---

Returns

The texture that was loaded

6.1.3.16 PrepareDraw()

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

This is automatically called once at the beginning of your [BIWindow3D::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 [BIWindow3D::FrameDraw](#) method. Only in the first call should `firstCallInDraw` be true, and in any subsequent calls it should be false.

Parameters

<i>firstCallInDraw</i>	True indicates this method should also sleep in order to obey FramePeriod.
------------------------	--

6.1.3.17 ResetCamera()

```
void Blotch.BIGraphicsDeviceManager.ResetCamera ( )
```

Sets [Eye](#), [LookAt](#), etc. back to default starting position.

6.1.3.18 SetCameraRollToZero()

```
void Blotch.BIGraphicsDeviceManager.SetCameraRollToZero ( )
```

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

6.1.3.19 SetCameraToSprite()

```
void Blotch.BIGraphicsDeviceManager.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

<i>sprite</i>	The sprite that the camera should be connected to
---------------	---

6.1.3.20 SetSpriteToCamera()

```
void Blotch.BIGraphicsDeviceManager.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

<i>sprite</i>	The sprite that should be connected to the camera
---------------	---

6.1.3.21 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

<i>text</i>	The text to write to the texture
<i>font</i>	Font to use
<i>color</i>	If specified, color of the text. (Default is white)
<i>backColor</i>	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!

6.1.4 Member Data Documentation

6.1.4.1 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 [BISprite](#). Both diffuse and ambient light illuminates the model's Color. See the [BISprite::Color](#) member.

6.1.4.2 Aspect

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

The aspect ratio.

6.1.4.3 AutoRotate

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

How fast [DoDefaultGui](#) should auto-rotate the scene.

6.1.4.4 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.

6.1.4.5 CameraUp

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

Camera Up vector. Initially set to +Z. [ResetCamera](#) and [SetCameraToSprite](#) updates this.

6.1.4.6 ClearColor

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

The background color.

6.1.4.7 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

6.1.4.8 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

6.1.4.9 DepthStencilStateDisabled

DepthStencilState Blotch.BlGraphicsDeviceManager.DepthStencilStateDisabled

Initial value:

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

Assign DepthStencilState to this to disable depth buffering

6.1.4.10 DepthStencilStateEnabled

DepthStencilState Blotch.BlGraphicsDeviceManager.DepthStencilStateEnabled

Initial value:

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

Assign DepthStencilState to this to enable depth buffering

6.1.4.11 FarClip

double Blotch.BlGraphicsDeviceManager.FarClip = 0

The far clipping plane, or 0 = autoclip.

6.1.4.12 FogColor

Vector3 Blotch.BlGraphicsDeviceManager.FogColor = null

If not null, color of fog.

6.1.4.13 fogEnd

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

How far away fog ends. See [FogColor](#).

6.1.4.14 fogStart

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

How far away fog starts. See [FogColor](#).

6.1.4.15 FramePeriod

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

How much time between consecutive frames.

6.1.4.16 IsDisposed

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

Set when the object is Disposed.

6.1.4.17 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.

6.1.4.18 NearClip

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

The near clipping plane, or 0 = autclip.

6.1.4.19 Projection

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

The Projection matrix. Normally you would use the higher-level functions [Zoom](#), [Aspect](#), [NearClip](#), or [FarClip](#) instead of changing this directly.

6.1.4.20 SpriteBatch

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

A [SpriteBatch](#) for use by certain text and texture drawing methods.

6.1.4.21 TargetEye

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

The point that [Eye](#) migrates to, according to [CameraSpeed](#). See [Eye](#) for more information.

6.1.4.22 TargetLookAt

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

The point that [LookAt](#) migrates to, according to [CameraSpeed](#). See [LookAt](#) for more information.

6.1.4.23 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](#) instead of changing this directly.

6.1.4.24 Window

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

The [BlWindow3D](#) associated with this object.

6.1.4.25 Zoom

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

The field of view, in degrees.

6.1.5 Property Documentation

6.1.5.1 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](#).

6.1.5.2 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](#).

6.1.5.3 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](#).

6.1.5.4 CameraRight

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

Camera Right vector. Writes to [Eye](#) and [LookAt](#), and calls to [SetCameraToSprite](#) cause this to be updated.

6.1.5.5 CurrentAspect

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

Current aspect ratio. Same as [Aspect](#) unless [Aspect](#)==0.

6.1.5.6 CurrentFarClip

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

Current value of far clipping plane. See [FarClip](#).

6.1.5.7 CurrentNearClip

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

Current value of near clipping plane. See [NearClip](#).

6.1.5.8 Eye

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

The current camera position. Note: To change the camera position, set [TargetEye](#). Also see [CameraSpeed](#).

6.1.5.9 LookAt

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

The current camera LookAt position. Note: To change the camera LookAt, set [TargetLookAt](#). Also see [CameraSpeed](#).

6.1.5.10 MaxCamDistance

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

Distance to the farthest sprite, plus its radius. Note this is set to a very small number by [PrepareDraw](#), and then as [BIWindow3D::FrameDraw](#) is called it is set more reasonably.

6.1.5.11 MinCamDistance

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

Distance to the nearest sprite, less its radius. Note this is set to a very large number by [PrepareDraw](#), and then as [BIWindow3D::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/BIGraphicsDeviceManager.cs

6.2 Blotch.BIGuiControl Class Reference

A 2D GUI control. To create a GUI control: instantiate one of these, set its initial Texture (remember to create it in the 3D thread context), window position, and delegate, and then add it to [BIWindow3D::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 [BIWindow3D::FrameProc](#) method. You can use [BIGraphicsDeviceManager::TextToTexture](#) to create a textual textures, or just load a texture from a content file. Remember to Dispose textures when you are done with them.

Public Member Functions

- delegate void [OnMouseChangeDelegate](#) ([BIGuiControl](#) guiCtrl)

Delegates for a [BIGuiControl](#) are of this type

- **BIGuiControl** ([BIWindow3D](#) window)
- bool [HandleInput](#) ()

Periodically called by [BIWindow3D](#). 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 in the [BIWindow3D](#) of this control

- [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.

- [BIWindow3D](#) [Window](#) = null

The window this [BIGuiControl](#) is in.

6.2.1 Detailed Description

A 2D GUI control. To create a GUI control: instantiate one of these, set its initial Texture (remember to create it in the 3D thread context), window position, and delegate, and then add it to [BIWindow3D::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 [BIWindow3D::FrameProc](#) method. You can use [BIGraphicsDeviceManager::TextToTexture](#) to create a textual textures, or just load a texture from a content file. Remember to Dispose textures when you are done with them.

6.2.2 Member Function Documentation

6.2.2.1 HandleInput()

```
bool Blotch.BIGuiControl.HandleInput ( )
```

Periodically called by [BIWindow3D](#). You shouldn't need to call this.

Returns

True if mouse is over any control, false otherwise.

6.2.2.2 OnMouseChangeDelegate()

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

Delegates for a [BIGuiControl](#) are of this type

Parameters

<i>guiCtrl</i>	
----------------	--

6.2.3 Member Data Documentation

6.2.3.1 OnMouseOver

```
OnMouseChangeDelegate Blotch.BIGuiControl.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`).

6.2.3.2 Position

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

The pixel position in the [BlWindow3D](#) of this control

6.2.3.3 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.

6.2.3.4 Texture

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

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

6.2.3.5 Window

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

The window this [BlGuiControl](#) is in.

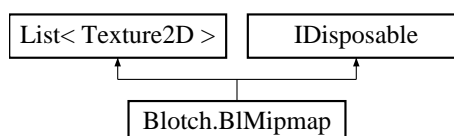
The documentation for this class was generated from the following file:

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

6.3 Blotch.BIMipmap Class Reference

A mipmap of textures for a given [BlSprite](#). You could load this from an image file and then 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, only one resolution texture is used at time.

Inheritance diagram for Blotch.BIMipmap:



Public Member Functions

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

Creates the mipmaps.

- 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

- bool **IsDisposed** = false

Set when the object is Disposed.

6.3.1 Detailed Description

A mipmap of textures for a given **BlSprite**. You could load this from an image file and then 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, only one resolution texture is used at time.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 BIMipmap()

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

Creates the mipmaps.

Parameters

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

6.3.3 Member Function Documentation

6.3.3.1 Dispose()

```
void Blotch.BIMipmap.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.

6.3.4 Member Data Documentation

6.3.4.1 IsDisposed

```
bool Blotch.BIMipmap.IsDisposed = false
```

Set when the object is Disposed.

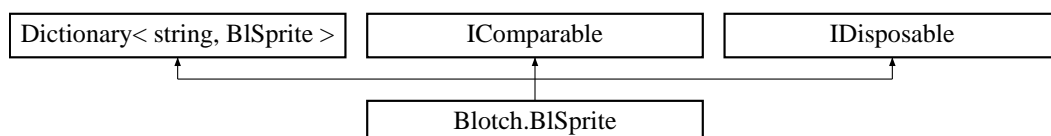
The documentation for this class was generated from the following file:

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

6.4 Blotch.BISprite Class Reference

A [BISprite](#) is a single 3D object. Each sprite can also hold any number of subsprites, so you can make a sprite tree (a scene graph). In that case the 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.BISprite:



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 [SetMeshEffectCmd](#) { [SetMeshEffectCmd.Continue](#), [SetMeshEffectCmd.Abort](#), [SetMeshEffectCmd.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](#) ([BISprite](#) sprite)
See [#FrameProc](#)
- void [ExecuteFrameProc](#) ()
Execute the [FrameProc](#), if it was specified in the [BISprite](#) constructor. (Normally you wouldn't need to call this because its automatically called by the [BIWindow](#).)
- delegate [PreDrawCmd](#) [PreDrawType](#) ([BISprite](#) sprite)
See [PreDraw](#)
- delegate [PreSubspritesCmd](#) [PreSubspritesType](#) ([BISprite](#) sprite)
See [PreSubsprites](#)
- delegate [SetMeshEffectCmd](#) [SetMeshEffectType](#) ([BISprite](#) sprite, [ModelMesh](#) mesh)
See [SetMeshEffect](#)
- delegate [PreLocalCmd](#) [PreLocalType](#) ([BISprite](#) sprite)
See [PreLocal](#)
- delegate void [DrawCleanupType](#) ([BISprite](#) sprite)
See [DrawCleanup](#)
- [BISprite](#) ([BISGraphicsDeviceManager](#) graphicsIn, string name, [FrameProcType](#) frameProc=null)
Constructs a sprite
- void [Add](#) ([BISprite](#) s)
Add a subsprite. (A [BISprite](#) inherits from a Dictionary of [BISprites](#). 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 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](#)< [BISprite](#) > [GetRayIntersections](#) ([Ray](#) ray, ulong flags=0xFFFFFFFFFFFFFFFF, [List](#)< [BISprite](#) > 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](#) [GetMipmapLod](#) ()
- 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 `BIDebug.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 [SetMeshEffect](#)) 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.

- 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 triangle list (`VertexPositionNormalTexture[]`), or null (indicating nothing should be drawn). 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).

- [BIMipmap Mipmap](#) = null

Mipmap textures to apply to the model. These work the same as LODs (see LODs for more information). The texture used 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 one is used. So, for example, you can specify only one texture to be used as all mipmaps if you like. Note that for a texture to display, the model must include texture coordinates. It must also include normals if lighting other than 'emissive' is desired. 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. These are NOT disposed when the sprite is disposed, so a given [BIMipmap](#) 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).

- BoundingBoxSphere [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: $2 * \text{mag}^2 - 1 / \text{mag}^2$. 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: $2 * \text{mag}^2 - 1 / \text{mag}^2$. 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: $2 * \text{mag}^2 - 1 / \text{mag}^2$. 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 [SetMeshEffect](#)) 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](#).

- [BGraphicsDeviceManager 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 [SetMeshEffect](#)).

- 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 [SetMeshEffect](#)).

- bool [IncludeInAutoClipping](#) = true

Whether to use depth testing, and whether 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 [SetMeshEffect](#)).

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

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, .2f)

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](#) = 8

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 [BISprite](#) 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 [BISprite](#) field.

- [SetMeshEffectType SetMeshEffect](#) = null

If this not null, then the [Draw](#) method executes this delegate for each model mesh instead of setting the mesh effect to [BasicEffect](#). In that case some effect must be set for the mesh. See the [SpriteAlphaTexture](#) example. The return value from the delegate indicates whether to draw the mesh, skip drawing the mesh, or abort the Draw method entirely. Note: This works only when for LOD elements that are models. When an LOD element is a vertex list, it always uses the [BasicEffect](#).

- [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 [BISprite](#) 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 [BISprite](#)

- 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.

6.4.1 Detailed Description

A [BISprite](#) is a single 3D object. Each sprite can also hold any number of subsprites, so you can make a sprite tree (a scene graph). In that case the 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.

6.4.2 Member Enumeration Documentation

6.4.2.1 PreDrawCmd

```
enum Blotch.BISprite.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.

6.4.2.2 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

6.4.2.3 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

6.4.2.4 SetMeshEffectCmd

```
enum Blotch.BlSprite.SetMeshEffectCmd [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

6.4.3 Constructor & Destructor Documentation

6.4.3.1 BISprite()

```
Blotch.BISprite.BISprite (
    BGraphicsDeviceManager graphicsIn,
    string name,
    FrameProcType frameProc = null )
```

Constructs a sprite

Parameters

<i>graphicsIn</i>	The BGraphicsDeviceManager that operates on this sprite
<i>name</i>	The name of the sprite (must be unique among other sprites in the same subsprite list)
<i>frameProc</i>	The delegate to run every frame

6.4.4 Member Function Documentation

6.4.4.1 Add()

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

Add a subsprite. (A [BISprite](#) inherits from a Dictionary of BISprites. This wrapper method to the dictionary's Add method simply adds the sprite where the key is the sprite's [Name](#).)

Parameters

<i>s</i>	
----------	--

6.4.4.2 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

<i>obj</i>	
------------	--

Returns

6.4.4.3 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.

6.4.4.4 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

<i>ray</i>	
------------	--

Returns

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

6.4.4.5 Draw()

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

Draws the sprite and the subsprites.

Parameters

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

6.4.4.6 DrawCleanupType()

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

See [DrawCleanup](#)

Parameters

<i>sprite</i>	
---------------	--

6.4.4.7 ExecuteFrameProc()

```
void Blotch.BISprite.ExecuteFrameProc ( )
```

Execute the FrameProc, if it was specified in the [BISprite](#) constructor. (Normally you wouldn't need to call this because its automatically called by the BIWindow.)

6.4.4.8 FrameProcType()

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

See [#FrameProc](#)

Parameters

<i>sprite</i>	
---------------	--

6.4.4.9 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

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

Returns

A list of subsprites that the ray hit

6.4.4.10 GetViewCoords()

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

Returns the current 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

6.4.4.11 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

<i>point1</i>	
<i>point2</i>	
<i>nearPoint</i>	

Returns

Point on the line nearest to nearPoint

6.4.4.12 PreDrawType()

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

See [PreDraw](#)

Parameters

<i>sprite</i>	
---------------	--

Returns**6.4.4.13 PreLocalType()**

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

See [PreLocal](#)

Parameters

<i>sprite</i>	
---------------	--

Returns**6.4.4.14 PreSubspritesType()**

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

See [PreSubsprites](#)

Parameters

<i>sprite</i>	
---------------	--

Returns

6.4.4.15 SetAllMaterialBlack()

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

Sets all material colors to black.

6.4.4.16 SetMeshEffectType()

```
delegate SetMeshEffectCmd Blotch.BlSprite.SetMeshEffectType (
    BlSprite sprite,
    ModelMesh mesh )
```

See [SetMeshEffect](#)

Parameters

<i>sprite</i>	
<i>mesh</i>	

Returns

6.4.5 Member Data Documentation

6.4.5.1 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 [SetMeshEffect](#)) may need to change it from within the [Draw](#) method. This is the matrix that is also passed to subsprites as their 'world' matrix.

6.4.5.2 BoundSphere

```
BoundingBox 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.

6.4.5.3 Color

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

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

6.4.5.4 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.

6.4.5.5 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: $2 * \text{mag}^2 - 1 / \text{mag}^2$. 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](#).

6.4.5.6 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: $2 * \text{mag}^2 - 1 / \text{mag}^2$. 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](#).

6.4.5.7 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: $2 * \text{mag}^2 - 1 / \text{mag}^2$. 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](#).

6.4.5.8 DrawCleanup

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

If not null, [Draw](#) method calls this at the end.

6.4.5.9 EmissiveColor

```
Vector3 Blotch.BlSprite.EmissiveColor = new Vector3(.1f, .1f, .2f)
```

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

6.4.5.10 Flags

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

The Flags field can be used by callbacks of [Draw](#) ([PreDraw](#), [PreSubsprites](#), [PreLocal](#), and [SetMeshEffect](#)) 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.

6.4.5.11 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 [SetMeshEffect](#)).

6.4.5.12 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 [SetMeshEffect](#)).

6.4.5.13 IncludeInAutoClipping

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

Whether to use depth testing, and whether to participate in autoclipping calculations when they are enabled.

6.4.5.14 IsDisposed

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

Set when the object is Disposed.

6.4.5.15 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 [SetMeshEffect](#)).

6.4.5.16 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 triangle list ([VertexPositionNormalTexture\[\]](#)), or null (indicating nothing should be drawn). 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.

6.4.5.17 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).

6.4.5.18 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](#).

6.4.5.19 Mipmap

```
BlMipmap Blotch.BlSprite.Mipmap = null
```

Mipmap textures to apply to the model. These work the same as LODs (see LODs for more information). The texture used 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 one is used. So, for example, you can specify only one texture to be used as all mipmaps if you like. Note that for a texture to display, the model must include texture coordinates. It must also include normals if lighting other than 'emissive' is desired. 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. These are NOT disposed when the sprite is disposed, so a given [BlMipmap](#) may be assigned to multiple sprites.

6.4.5.20 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).

6.4.5.21 Name

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

The name of the [BlSprite](#)

6.4.5.22 PreDraw

```
PreDrawType Blotch.BISprite.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 [BISprite](#) field, and/or control the further execution of the Draw method.

6.4.5.23 PreLocal

```
PreLocalType Blotch.BISprite.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 [BISprite](#) field, and/or abort the [Draw](#) method.

6.4.5.24 PreSubsprites

```
PreSubspritesType Blotch.BISprite.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 [BISprite](#) field.

6.4.5.25 SetMeshEffect

```
SetMeshEffectType Blotch.BISprite.SetMeshEffect = null
```

If this not null, then the [Draw](#) method executes this delegate for each model mesh instead of setting the mesh effect to BasicEffect. In that case some effect must be set for the mesh. See the SpriteAlphaTexture example. The return value from the delegate indicates whether to draw the mesh, skip drawing the mesh, or abort the Draw method entirely. Note: This works only when for LOD elements that are models. When an LOD element is a vertex list, it always uses the BasicEffect.

6.4.5.26 SpecularColor

```
Vector3 Blotch.BISprite.SpecularColor = null
```

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

6.4.5.27 SpecularPower

```
float Blotch.BlSprite.SpecularPower = 8
```

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

6.4.5.28 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](#).

6.4.6 Property Documentation

6.4.6.1 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)

6.4.6.2 CamDistance

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

Distance to the camera.

6.4.6.3 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.

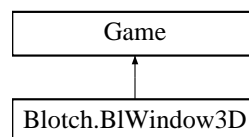
The documentation for this class was generated from the following file:

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

6.5 Blotch.BIWindow3D Class Reference

To make a 3D window, you must derive a class from [BIWindow3D](#) 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.BIWindow3D:



Public Member Functions

- delegate void [Command](#) ([BIWindow3D](#) win)
See [EnqueueCommand](#), [EnqueueCommandBlocking](#), and [BIWindow3D](#) for more info
- [BIWindow3D](#) ()
See [BIWindow3D](#) 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 [BISprites](#). You can also use this for general thread safety of various operations. This method does not block. Also see [BIWindow3D](#) 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 [BISprites](#). You can also use this for general thread safety of various operations. This method blocks until the command has executed. Also see [BIWindow3D](#) and the (non-blocking) [EnqueueCommand](#) for more details.
- void [FrameProcSpritesAdd](#) ([BISprite](#) s)
Used internally
- void [FrameProcSpritesRemove](#) ([BISprite](#) 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 [BIDebug.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

- [BIGraphicsDeviceManager](#) Graphics
The [BIGraphicsDeviceManager](#) associated with this window. This is automatically created when you create the [BIWindow3D](#).
- ConcurrentDictionary< string, [BIGuiControl](#) > [GuiControls](#) = new ConcurrentDictionary<string, [BIGuiControl](#)>()
The GUI controls for this window. See [BIGuiControl](#) 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 [BIWindow3D](#) for details.
- override void [Update](#) (GameTime timeInfo)
Used internally, Do NOT override. Use FrameProc instead.
- virtual void [FrameProc](#) (GameTime timeInfo)
See [BIWindow3D](#) for details.
- override void [Draw](#) (GameTime timeInfo)
Used internally, Do NOT override. Use FrameDraw instead.
- virtual void [FrameDraw](#) (GameTime timeInfo)
See [BIWindow3D](#) for details.

6.5.1 Detailed Description

To make a 3D window, you must derive a class from [BIWindow3D](#) 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](#).

6.5.2 Constructor & Destructor Documentation

6.5.2.1 BIWindow3D()

```
Blotch.BIWindow3D.BIWindow3D ( )
```

See [BIWindow3D](#) for details.

6.5.3 Member Function Documentation

6.5.3.1 Command()

```
delegate void Blotch.BIWindow3D.Command (
    BIWindow3D win )
```

See [EnqueueCommand](#), [EnqueueCommandBlocking](#), and [BIWindow3D](#) for more info

Parameters

<i>win</i>	The BIWindow3D object
------------	---------------------------------------

6.5.3.2 Dispose()

```
new void Blotch.BIWindow3D.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 `BIDebug.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.

6.5.3.3 Draw()

```
override void Blotch.BIWindow3D.Draw (
    GameTime timeInfo ) [protected]
```

Used internally, Do NOT override. Use `FrameDraw` instead.

Parameters

<i>timeInfo</i>	
-----------------	--

6.5.3.4 EnqueueCommand()

```
void Blotch.BIWindow3D.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 `BISprites`. You can also use this for general thread safety of various operations. This method does not block. Also see [BIWindow3D](#) and the (blocking) [EnqueueCommandBlocking](#) for more details.

Parameters

<i>cmd</i>	
------------	--

6.5.3.5 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

<i>cmd</i>	
------------	--

6.5.3.6 FrameDraw()

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

See [BlWindow3D](#) for details.

Parameters

<i>timeInfo</i>	
-----------------	--

6.5.3.7 FrameProc()

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

See [BlWindow3D](#) for details.

Parameters

<i>timeInfo</i>	
-----------------	--

6.5.3.8 FrameProcSpritesAdd()

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

Used internally

Parameters

<i>s</i>	
----------	--

6.5.3.9 FrameProcSpritesRemove()

```
void Blotch.BIWindow3D.FrameProcSpritesRemove (
    BlSprite s )
```

Used internally

Parameters

<i>s</i>	
----------	--

6.5.3.10 Initialize()

```
override void Blotch.BIWindow3D.Initialize ( ) [protected]
```

Used internally, Do NOT override. Use Setup instead.

6.5.3.11 LoadContent()

```
override void Blotch.BIWindow3D.LoadContent ( ) [protected]
```

Used internally, Do NOT override. Use Setup instead.

6.5.3.12 Setup()

```
virtual void Blotch.BIWindow3D.Setup ( ) [protected], [virtual]
```

Override this and put all initialization and global content creation code in it. See [BIWindow3D](#) for details.

6.5.3.13 Update()

```
override void Blotch.BIWindow3D.Update (
    GameTime timeInfo ) [protected]
```

Used internally, Do NOT override. Use FrameProc instead.

Parameters

<i>timeInfo</i>	
-----------------	--

6.5.4 Member Data Documentation

6.5.4.1 Graphics

`BlGraphicsDeviceManager` Blotch.BlWindow3D.Graphics

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

6.5.4.2 GuiControls

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

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

6.5.4.3 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

6.6 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)

6.6.1 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