

# Namespace Bliss.CSharp

## Classes

[Disposable](#)

[GlobalResource](#)

# Class Disposable

Namespace: [Bliss.CSharp](#)

Assembly: Bliss.dll

```
public abstract class Disposable : IDisposable
```

## Inheritance

[object](#) ← Disposable

## Implements

[IDisposable](#)

## Derived

[Effect](#), [Font](#), [Mesh](#), [Model](#), [SimpleBufferLayout](#), [SimpleBuffer<T>](#), [SimplePipeline](#), [SimpleTextureLayout](#), [PrimitiveBatch](#), [SpriteBatch](#), [FullScreenRenderPass](#), [ImmediateRenderer](#), [Sdl3InputContext](#), [Sdl3Gamepad](#), [Sdl3Cursor](#), [Cubemap](#), [RenderTexture2D](#), [Texture2D](#), [Sdl3Window](#)

## Inherited Members

[object.Equals\(object\)](#), [object.Equals\(object, object\)](#), [object.GetHashCode\(\)](#), [object.GetType\(\)](#), [object.MemberwiseClone\(\)](#), [object.ReferenceEquals\(object, object\)](#), [object.ToString\(\)](#)

# Properties

## HasDisposed

Indicates whether the object has been disposed.

```
public bool HasDisposed { get; }
```

## Property Value

[bool](#)

True if the object has been disposed; otherwise, false.

# Methods

## Dispose()

Disposes of the object, allowing for proper resource cleanup and finalization.

```
public void Dispose()
```

## Dispose(bool)

Disposes of the object and releases associated resources.

```
protected abstract void Dispose(bool disposing)
```

Parameters

`disposing` [bool](#)

True if called from user code; false if called from a finalizer.

## ~Disposable()

Disposes of the object, allowing for proper resource cleanup and finalization.

```
protected ~Disposable()
```

## ThrowIfDisposed()

Throws an exception if the object has been disposed, indicating that it is no longer usable.

```
protected void ThrowIfDisposed()
```

# Class GlobalResource

Namespace: [Bliss.CSharp](#)

Assembly: Bliss.dll

```
public static class GlobalResource
```

## Inheritance

[object](#) ← GlobalResource

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Properties

### BufferLayouts

Stores a collection of globally accessible buffer layouts used in rendering pipelines.

```
public static List<SimpleBufferLayout> BufferLayouts { get; }
```

#### Property Value

[List](#) <[SimpleBufferLayout](#)>

### DefaultFullScreenRenderPassEffect

Gets the default [Effect](#) used for full-screen render passes.

```
public static Effect DefaultFullScreenRenderPassEffect { get; }
```

#### Property Value

[Effect](#)

## DefaultImmediateRendererEffect

Gets the default [Effect](#) used for immediate mode rendering operations.

```
public static Effect DefaultImmediateRendererEffect { get; }
```

Property Value

[Effect](#)

## DefaultImmediateRendererTexture

The default [Texture2D](#) used for immediate mode rendering.

```
public static Texture2D DefaultImmediateRendererTexture { get; }
```

Property Value

[Texture2D](#)

## DefaultModelEffect

The default [Effect](#) used for rendering 3D models.

```
public static Effect DefaultModelEffect { get; }
```

Property Value

[Effect](#)

## DefaultModelTexture

The default [Texture2D](#) used for rendering 3D models.

```
public static Texture2D DefaultModelTexture { get; }
```

Property Value

[Texture2D](#)

## DefaultPrimitiveEffect

Gets the [Effect](#) used for rendering primitive shapes.

```
public static Effect DefaultPrimitiveEffect { get; }
```

Property Value

[Effect](#)

## DefaultSpriteEffect

Gets the default [Effect](#) used for rendering sprites.

```
public static Effect DefaultSpriteEffect { get; }
```

Property Value

[Effect](#)

## GraphicsDevice

Provides access to the global graphics device used for rendering operations.

```
public static GraphicsDevice GraphicsDevice { get; }
```

Property Value

[GraphicsDevice](#) ↗

## TextureLayouts

Maintains a collection of texture layouts used for configuring and managing texture bindings in graphics rendering operations.

```
public static List<SimpleTextureLayout> TextureLayouts { get; }
```

## Property Value

[List](#) <[SimpleTextureLayout](#)>

## Methods

### CreateBufferLayout(string, SimpleBufferType, ShaderStages)

Creates a new buffer layout and adds it to the global list of buffer layouts.

```
public static SimpleBufferLayout CreateBufferLayout(string name, SimpleBufferType  
bufferType, ShaderStages stages)
```

#### Parameters

**name** [string](#)

The name of the buffer layout to create.

**bufferType** [SimpleBufferType](#)

The type of buffer being created, such as uniform or structured.

**stages** [ShaderStages](#)

The shader stages where the buffer will be used.

#### Returns

[SimpleBufferLayout](#)

The created [SimpleBufferLayout](#).

### CreateTextureLayout(string)

Creates a new texture layout and adds it to the global collection of texture layouts.

```
public static SimpleTextureLayout CreateTextureLayout(string name)
```

## Parameters

`name` [string](#)

The name of the texture layout.

## Returns

[SimpleTextureLayout](#)

A new `SimpleTextureLayout` instance initialized with the specified name.

## Destroy()

Releases and disposes of all global resources.

```
public static void Destroy()
```

## Init(GraphicsDevice)

Initializes global resources.

```
public static void Init(GraphicsDevice graphicsDevice)
```

## Parameters

`graphicsDevice` [GraphicsDevice](#)

The graphics device to be used for resource creation and rendering.

# Namespace Bliss.CSharp.Audio

## Classes

[Vector3FExtensions](#)

# Class Vector3FExtensions

Namespace: [Bliss.CSharp.Audio](#)

Assembly: Bliss.dll

```
public static class Vector3FExtensions
```

## Inheritance

[object](#) ← Vector3FExtensions

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Methods

### ToVector3(Vector3f)

Converts a MiniAudioEx.Vector3f instance to a [Vector3](#) instance.

```
public static Vector3 ToVector3(this Vector3f v)
```

#### Parameters

v Vector3f

The MiniAudioEx.Vector3f instance to convert.

#### Returns

[Vector3](#)

A [Vector3](#) instance corresponding to the given MiniAudioEx.Vector3f.

### ToVector3F(Vector3)

Converts a [Vector3](#) instance to a MiniAudioEx.Vector3f instance.

```
public static Vector3f ToVector3F(this Vector3 v)
```

## Parameters

v [Vector3](#)

The [Vector3](#) instance to convert.

## Returns

Vector3f

A MiniAudioEx.Vector3f instance corresponding to the given [Vector3](#).

# Namespace Bliss.CSharp.Camera

## Interfaces

[ICam](#)

# Interface ICam

Namespace: [Bliss.CSharp.Camera](#)

Assembly: Bliss.dll

```
public interface ICam
```

## Methods

### Begin(CommandList)

Initializes the camera's usage in the current frame and sets it as the active camera.

```
void Begin(CommandList commandList)
```

#### Parameters

`commandList` [CommandList](#)

The command list to begin the camera's drawing operations.

### End()

Concludes the camera's operations for the current frame.

```
void End()
```

### Resize(uint, uint)

Resizes the viewport and updates the aspect ratio based on the given width and height.

```
void Resize(uint width, uint height)
```

#### Parameters

**width** [uint](#)

The new width of the viewport.

**height** [uint](#)

The new height of the viewport.

## Update(double)

Updates the camera's state, recalculating its parameters as needed.

**void** [Update](#)(**double** timeStep)

### Parameters

**timeStep** [double](#)

# Namespace Bliss.CSharp.Camera.Dim2

## Classes

[Cam2D](#)

## Enums

[CameraFollowMode](#)

# Class Cam2D

Namespace: [Bliss.CSharp.Camera.Dim2](#)

Assembly: Bliss.dll

```
public class Cam2D : ICam
```

## Inheritance

[object](#) ← Cam2D

## Implements

[ICam](#)

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

Cam2D(Vector2, Vector2, Rectangle, CameraFollowMode,  
Vector2?, float, float)

Initializes a new instance of the [Cam2D](#) class.

```
public Cam2D(Vector2 position, Vector2 target, Rectangle size, CameraFollowMode mode,  
Vector2? offset = null, float rotation = 0, float zoom = 1)
```

## Parameters

**position** [Vector2](#)

The initial position of the camera.

**target** [Vector2](#)

The target position the camera follows.

**size** [Rectangle](#)

The dimensions of the camera's viewport.

#### mode [CameraFollowMode](#)

The camera follow mode determining how the camera tracks the target.

#### offset [Vector2](#)?

Optional offset applied to the camera's position. Defaults to (0,0).

#### rotation [float](#)

The initial rotation of the camera in degrees.

#### zoom [float](#)

The zoom level of the camera. A value of 1.0 represents no zoom.

## Fields

### FractionFollowSpeed

Represents the fraction of the distance to the target that the camera covers per update cycle, used to determine the speed of the camera's follow movement.

```
public float FractionFollowSpeed
```

### Field Value

#### [float](#)

### MinFollowEffectLength

Represents the minimum distance at which the follow effect is activated.

```
public float MinFollowEffectLength
```

### Field Value

#### [float](#)

## MinFollowSpeed

Represents the minimum speed at which the camera follows its target.

```
public float MinFollowSpeed
```

### Field Value

[float](#)

## Mode

Defines the camera's follow mode, determining how the camera follows its target.

```
public CameraFollowMode Mode
```

### Field Value

[CameraFollowMode](#)

## Offset

Gets or sets the offset position of the camera.

```
public Vector2 Offset
```

### Field Value

[Vector2](#)

## Position

Gets or sets the position of the camera in 2D space.

```
public Vector2 Position
```

## Field Value

[Vector2](#)

## Rotation

Gets or sets the rotation angle of the camera, in degrees.

```
public float Rotation
```

## Field Value

[float](#)

## Target

Gets or sets the target position of the camera.

```
public Vector2 Target
```

## Field Value

[Vector2](#)

## Zoom

Gets or sets the zoom level of the camera.

```
public float Zoom
```

## Field Value

[float](#)

# Properties

## ActiveCamera

Gets or sets the active camera instance.

```
public static Cam2D? ActiveCamera { get; }
```

### Property Value

[Cam2D](#)

## Size

Gets the current viewport settings of the camera.

```
public Rectangle Size { get; }
```

### Property Value

[Rectangle](#)

# Methods

## Begin(CommandList)

Initializes the camera's usage in the current frame and sets it as the active camera.

```
public void Begin(CommandList commandList)
```

### Parameters

**commandList** [CommandList](#) ↗

The command list to begin the camera's drawing operations.

## End()

Ends the current camera session, setting the active camera to null.

```
public void End()
```

## GetScreenToWorld(Vector2)

Converts a screen position to a world position based on the camera's current view matrix.

```
public Vector2 GetScreenToWorld(Vector2 position)
```

Parameters

**position** [Vector2](#)

The screen position to be converted.

Returns

[Vector2](#)

The corresponding world position.

## GetView()

Retrieves the current view matrix of the camera.

```
public Matrix4x4 GetView()
```

Returns

[Matrix4x4](#)

The [Matrix4x4](#) representing the camera's view matrix.

## GetWorldToScreen(Vector2)

Converts a given world position to screen coordinates based on the camera's current view matrix.

```
public Vector2 GetWorldToScreen(Vector2 position)
```

Parameters

**position** [Vector2](#)

The world position to convert.

Returns

[Vector2](#)

A [Vector2](#) representing the screen coordinates corresponding to the given world position.

## Resize(uint, uint)

Resizes the camera's viewport to the specified width and height.

```
public void Resize(uint width, uint height)
```

Parameters

**width** [uint](#)

The new width of the viewport.

**height** [uint](#)

The new height of the viewport.

## Update(double)

Updates the camera's state based on the elapsed time and current mode.

```
public void Update(double timeStep)
```

## Parameters

### **timeStep** [double ↗](#)

The time elapsed since the last update, in seconds.

# Enum CameraFollowMode

Namespace: [Bliss.CSharp.Camera.Dim2](#)

Assembly: Bliss.dll

```
public enum CameraFollowMode
```

## Fields

**Custom = 0**

Custom follow behavior, allowing manual control of the camera's movement.

**FollowTarget = 1**

Directly follows the target position without any smoothing or delay.

**FollowTargetSmooth = 2**

Smoothly follows the target with a gradual transition, creating a smoother effect.

# Namespace Bliss.CSharp.Camera.Dim3

## Classes

[Cam3D](#)

## Enums

[CameraMode](#)

[ProjectionType](#)

# Class Cam3D

Namespace: [Bliss.CSharp.Camera.Dim3](#)

Assembly: Bliss.dll

```
public class Cam3D : ICam
```

## Inheritance

[object](#) ← Cam3D

## Implements

[ICam](#)

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

**Cam3D(Vector3, Vector3, float, Vector3?, ProjectionType, CameraMode, float, float, float)**

Initializes a new instance of the [Cam3D](#) class.

```
public Cam3D(Vector3 position, Vector3 target, float aspectRatio, Vector3? up = null,  
ProjectionType projectionType = ProjectionType.Perspective, CameraMode mode =  
CameraMode.Free, float fov = 70, float nearPlane = 0.01, float farPlane = 10000)
```

## Parameters

**position** [Vector3](#)

The camera position in world space.

**target** [Vector3](#)

The target point the camera is looking at.

**aspectRatio** [float](#)

The aspect ratio (width / height). Must be greater than zero.

### up [Vector3](#)?

The upward direction vector. Defaults to (0,1,0).

### projectionType [ProjectionType](#)

The type of projection (Perspective or Orthographic).

### mode [CameraMode](#)

The camera movement mode.

### fov [float](#)

The field of view in degrees. Should be between 1 and 179.

### nearPlane [float](#)

The near clipping plane. Must be positive and smaller than `farPlane`.

### farPlane [float](#)

The far clipping plane. Must be positive and greater than `nearPlane`.

## Fields

### FarPlane

Specifies the far clipping plane distance for the camera, determining the maximum depth at which objects are rendered in the 3D scene.

```
public float FarPlane
```

### Field Value

#### [float](#)

### Fov

Defines the field of view (FOV) angle for the camera, determining the extent of the observable world that is seen at any given moment.

```
public float Fov
```

Field Value

[float](#)

## Mode

Defines the current operational mode of the camera. This determines the behavior and control mechanics of the camera, such as whether it's in Free mode, Orbital mode, etc.

```
public CameraMode Mode
```

Field Value

[CameraMode](#)

## MouseSensitivity

Determines the sensitivity of the camera to mouse movements. This affects how fast the camera rotates or moves in response to mouse inputs. Higher values result in faster and more responsive camera movements.

```
public float MouseSensitivity
```

Field Value

[float](#)

## MovementSpeed

Specifies the speed at which the camera can move within the scene. Adjust this value to control how fast the camera translates in the 3D space. Typically used in conjunction with user input to navigate the

scene.

```
public float MovementSpeed
```

Field Value

[float](#)

## NearPlane

Specifies the near clipping plane distance for the camera, determining the minimum depth at which objects are rendered in the 3D scene.

```
public float NearPlane
```

Field Value

[float](#)

## OrbitalSpeed

Represents the speed at which the camera orbits around a target. Determines how quickly the camera moves when controlled to orbit in any direction. Typically used in orbital or tracking camera modes.

```
public float OrbitalSpeed
```

Field Value

[float](#)

## Position

Represents the position of the camera in a 3D space. This defines the location from which the camera is capturing its view.

```
public Vector3 Position
```

Field Value

[Vector3](#)

## ProjectionType

Represents the type of projection used by the camera, either Perspective or Orthographic, influencing how 3D scenes are rendered onto a 2D viewport.

`public ProjectionType ProjectionType`

Field Value

[ProjectionType](#)

## Target

Represents the focal point that the camera is aimed at in the 3D space. Determines the direction in which the camera is looking.

`public Vector3 Target`

Field Value

[Vector3](#)

## Up

Represents the upward direction vector for the camera, determining its orientation in the 3D space relative to its position and target.

`public Vector3 Up`

Field Value

[Vector3](#)

# Properties

## ActiveCamera

References the currently active instance of the Cam3D class. Used to determine which camera is currently rendering the scene. Can be accessed from other classes to retrieve camera-specific properties and methods.

```
public static Cam3D? ActiveCamera { get; }
```

### Property Value

[Cam3D](#)

## AspectRatio

Represents the ratio between the width and height of the camera's viewport. Used to adjust the projection matrix for rendering the 3D scene correctly on the screen.

```
public float AspectRatio { get; }
```

### Property Value

[float](#)

# Methods

## Begin(CommandList)

Sets the specified command list as the current command list for rendering operations and updates the camera's projection and view matrices. Also clears the depth stencil and sets this camera as the active camera.

```
public void Begin(CommandList commandList)
```

### Parameters

`commandList` [CommandList](#)

The command list to set as the current command list for rendering operations.

## End()

Ends the current 3D rendering session and deactivates the camera.

```
public void End()
```

## GetForward()

Computes and returns the forward direction vector based on the camera's position and target.

```
public Vector3 GetForward()
```

Returns

[Vector3](#)

A Vector3 representing the forward direction of the camera.

## GetFrustum()

Retrieves the camera frustum by extracting the frustum planes based on the current view and projection matrices.

```
public Frustum GetFrustum()
```

Returns

[Frustum](#)

The updated frustum containing the extracted planes.

## GetPitch()

Retrieves the current pitch angle of the camera.

```
public float GetPitch()
```

Returns

[float](#)

The current pitch angle in degrees.

## GetProjection()

Generates the projection matrix based on the current camera settings, such as projection type, field of view (FOV), aspect ratio, near plane, and far plane distances.

```
public Matrix4x4 GetProjection()
```

Returns

[Matrix4x4](#)

The calculated projection matrix.

## GetRight()

Computes and returns the right direction vector based on the camera's forward direction and up vector.

```
public Vector3 GetRight()
```

Returns

[Vector3](#)

A Vector3 representing the right direction of the camera.

## GetRoll()

Retrieves the roll component of the camera's rotation.

```
public float GetRoll()
```

Returns

[float](#)

The roll angle in degrees.

## GetRotation()

Retrieves the current rotation vector of the camera.

```
public Vector3 GetRotation()
```

Returns

[Vector3](#)

A Vector3 representing the current rotation of the camera.

## GetScreenToWorld(Vector2)

Converts a given screen position to world coordinates using the camera projection and view matrices.

```
public Vector3 GetScreenToWorld(Vector2 position)
```

Parameters

[position](#) [Vector2](#)

The screen position to transform.

Returns

[Vector3](#)

A [Vector3](#) representing the corresponding world coordinates.

## GetView()

Constructs and returns the view matrix for the camera based on its position, target, and up vector.

```
public Matrix4x4 GetView()
```

Returns

[Matrix4x4](#)

The view matrix of the camera.

## GetWorldToScreen(Vector3)

Converts a given world position to screen coordinates based on the camera's current view and projection matrices.

```
public Vector2 GetWorldToScreen(Vector3 position)
```

Parameters

**position** [Vector3](#)

The world position to convert.

Returns

[Vector2](#)

A [Vector2](#) representing the screen coordinates corresponding to the given world position.

## GetYaw()

Retrieves the current yaw (rotation around the Y-axis) of the camera.

```
public float GetYaw()
```

Returns

[float](#)

The yaw value in degrees.

## MoveForward(float, bool)

Moves the camera forward by a specified distance.

```
public void MoveForward(float distance, bool moveInWorldPlane)
```

Parameters

[distance](#) [float](#)

The distance by which to move the camera forward.

[moveInWorldPlane](#) [bool](#)

Determines whether to constrain movement to the world plane, ignoring the Y component.

## MoveRight(float, bool)

Moves the camera to the right by a specified distance, with an option to constrain the movement to the world plane.

```
public void MoveRight(float distance, bool moveInWorldPlane)
```

Parameters

[distance](#) [float](#)

The distance to move the camera to the right.

[moveInWorldPlane](#) [bool](#)

If set to `true`, the camera will move parallel to the ground plane and will not change its Y position.

## MoveToTarget(float)

Moves the camera towards or away from its target by modifying the distance between the camera's position and its target point.

```
public void MoveToTarget(float delta)
```

### Parameters

`delta` [float](#)

The amount by which to adjust the distance to the target. Positive values move the camera closer to the target, and negative values move it further away.

## MoveUp(float)

Moves the camera upward by a specified distance.

```
public void MoveUp(float distance)
```

### Parameters

`distance` [float](#)

The distance to move the camera upward.

## Resize(uint, uint)

Resizes the viewport and updates the aspect ratio based on the given width and height.

```
public void Resize(uint width, uint height)
```

### Parameters

`width` [uint](#)

The new width of the viewport.

### height [uint](#)

The new height of the viewport.

## SetPitch(float, bool)

Sets the pitch angle of the camera, rotating around the specified target if indicated.

```
public void SetPitch(float angle, bool rotateAroundTarget)
```

### Parameters

#### angle [float](#)

The desired pitch angle in degrees.

#### rotateAroundTarget [bool](#)

If true, rotates the camera around the target point; otherwise rotates around its own position.

## SetRoll(float)

Sets the roll angle of the camera by rotating around the forward axis.

```
public void SetRoll(float angle)
```

### Parameters

#### angle [float](#)

The angle in degrees by which to set the roll.

## SetYaw(float, bool)

Sets the yaw of the camera by rotating it around the target or adjusting the target position, based on the specified angle.

```
public void SetYaw(float angle, bool rotateAroundTarget)
```

## Parameters

angle [float](#)

The angle in degrees to rotate the camera.

rotateAroundTarget [bool](#)

A boolean indicating whether to rotate the camera around the target or adjust the target position.

## Update(double)

Updates the camera's state, recalculating its parameters as needed.

```
public void Update(double timeStep)
```

## Parameters

timeStep [double](#)

# Enum CameraMode

Namespace: [Bliss.CSharp.Camera.Dim3](#)

Assembly: Bliss.dll

```
public enum CameraMode
```

## Fields

**Custom** = 0

Custom mode, allowing for user-defined camera behavior and controls.

**FirstPerson** = 3

First-person mode, where the camera simulates the view of a character.

**Free** = 1

Free mode, where the camera can move freely in 3D space.

**Orbital** = 2

Orbital mode, where the camera orbits around a target point.

**ThirdPerson** = 4

Third-person mode, where the camera follows a character from a distance.

# Enum ProjectionType

Namespace: [Bliss.CSharp.Camera.Dim3](#)

Assembly: Bliss.dll

```
public enum ProjectionType
```

## Fields

Orthographic = 1

Perspective = 0

# Namespace Bliss.CSharp.Colors

## Structs

[Color](#)

# Struct Color

Namespace: [Bliss.CSharp.Colors](#)

Assembly: Bliss.dll

```
public readonly struct Color : IEquatable<Color>
```

Implements

[IEquatable](#)<[Color](#)>

Inherited Members

[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### Color(byte, byte, byte, byte)

Initializes a new instance of the [Color](#) class.

```
public Color(byte r, byte g, byte b, byte a)
```

Parameters

r [byte](#)

The red component.

g [byte](#)

The green component.

b [byte](#)

The blue component.

a [byte](#)

The alpha component.

# Color(RgbaFloat)

Represents a color with red, green, blue, and alpha components.

```
public Color(RgbaFloat rgbaFloat)
```

## Parameters

rgbaFloat [RgbaFloat](#)

## Fields

### A

Represents the alpha component of the color, indicating its transparency, in the range of 0 to 255.

```
public readonly byte A
```

### Field Value

[byte](#)

### B

Represents the blue component of the color in the range of 0 to 255.

```
public readonly byte B
```

### Field Value

[byte](#)

### Black

```
public static readonly Color Black
```

Field Value

[Color](#)

Blue

```
public static readonly Color Blue
```

Field Value

[Color](#)

Brown

```
public static readonly Color Brown
```

Field Value

[Color](#)

Cyan

```
public static readonly Color Cyan
```

Field Value

[Color](#)

DarkBlue

```
public static readonly Color DarkBlue
```

Field Value

[Color](#)

## DarkBrown

```
public static readonly Color DarkBrown
```

Field Value

[Color](#)

## DarkCyan

```
public static readonly Color DarkCyan
```

Field Value

[Color](#)

## DarkGray

```
public static readonly Color DarkGray
```

Field Value

[Color](#)

## DarkGreen

```
public static readonly Color DarkGreen
```

Field Value

[Color](#)

## DarkMagenta

```
public static readonly Color DarkMagenta
```

Field Value

[Color](#)

## DarkOrange

```
public static readonly Color DarkOrange
```

Field Value

[Color](#)

## DarkPink

```
public static readonly Color DarkPink
```

Field Value

[Color](#)

## DarkPurple

```
public static readonly Color DarkPurple
```

Field Value

[Color](#)

## DarkRed

```
public static readonly Color DarkRed
```

Field Value

[Color](#)

## DarkYellow

```
public static readonly Color DarkYellow
```

Field Value

[Color](#)

## G

Represents the green component of the color in the range of 0 to 255.

```
public readonly byte G
```

Field Value

[byte](#)

## Gray

```
public static readonly Color Gray
```

Field Value

[Color](#)

## Green

```
public static readonly Color Green
```

Field Value

[Color](#)

## LightBlue

```
public static readonly Color LightBlue
```

Field Value

[Color](#)

## LightBrown

```
public static readonly Color LightBrown
```

Field Value

[Color](#)

## LightCyan

```
public static readonly Color LightCyan
```

Field Value

[Color](#)

## LightGray

```
public static readonly Color LightGray
```

Field Value

[Color](#)

## LightGreen

```
public static readonly Color LightGreen
```

Field Value

[Color](#)

## LightMagenta

```
public static readonly Color LightMagenta
```

Field Value

[Color](#)

## LightOrange

```
public static readonly Color LightOrange
```

Field Value

[Color](#)

## LightPink

```
public static readonly Color LightPink
```

Field Value

[Color](#)

## LightPurple

```
public static readonly Color LightPurple
```

Field Value

[Color](#)

## LightRed

```
public static readonly Color LightRed
```

Field Value

[Color](#)

## LightYellow

```
public static readonly Color LightYellow
```

Field Value

[Color](#)

## Magenta

```
public static readonly Color Magenta
```

Field Value

[Color](#)

Orange

```
public static readonly Color Orange
```

Field Value

[Color](#)

Pink

```
public static readonly Color Pink
```

Field Value

[Color](#)

Purple

```
public static readonly Color Purple
```

Field Value

[Color](#)

R

Represents the red component of the color in the range of 0 to 255.

```
public readonly byte R
```

Field Value

[byte](#)

Red

```
public static readonly Color Red
```

Field Value

[Color](#)

White

```
public static readonly Color White
```

Field Value

[Color](#)

Yellow

```
public static readonly Color Yellow
```

Field Value

[Color](#)

## Methods

### Equals(Color)

Determines whether the current color object is equal to another color object.

```
public bool Equals(Color other)
```

Parameters

`other` [Color](#)

The color to compare to.

Returns

[bool](#)

True if the current color object is equal to the other color object; otherwise, false.

## Equals(object?)

Determines whether the current color object is equal to another color object.

```
public override bool Equals(object? obj)
```

Parameters

`obj` [object](#)

The color to compare to.

Returns

[bool](#)

True if the current color object is equal to the other color object; otherwise, false.

## GetHashCode()

Returns the hash code for this instance.

```
public override int GetHashCode()
```

Returns

[int](#)

The hash code for this instance.

## ToRgbaFloat()

Converts the color to an [RgbaFloat](#) value.

```
public RgbaFloat ToRgbaFloat()
```

Returns

[RgbaFloat](#)

A new instance of the [RgbaFloat](#) struct representing the color.

## ToRgbaFloatVec4()

Converts the color to a [Vector4](#) representation with each component normalized to the range of 0 to 1.

```
public Vector4 ToRgbaFloatVec4()
```

Returns

[Vector4](#)

A [Vector4](#) object representing the normalized RGBA components of the color.

## ToString()

Returns a string that represents the current color.

```
public override string ToString()
```

Returns

[string](#)

A string representation of the current color.

## ToVector4()

Converts the color to a Vector4 representation.

```
public Vector4 ToVector4()
```

Returns

[Vector4](#)

A Vector4 representing the color.

# Operators

## operator ==(Color, Color)

Determines whether two Color objects are equal.

```
public static bool operator ==(Color left, Color right)
```

Parameters

**left** [Color](#)

The first Color to compare.

**right** [Color](#)

The second Color to compare.

Returns

[bool](#)

`true` if the specified `Color` objects are equal; otherwise, `false`.

## operator !=(Color, Color)

Represents the equality operator for comparing two colors for equality.

```
public static bool operator !=(Color left, Color right)
```

Parameters

`left` [Color](#)

The first color to compare.

`right` [Color](#)

The second color to compare.

Returns

[bool](#)

`true` if the colors are equal; otherwise, `false`.

# Namespace Bliss.CSharp.Effects

## Classes

[Effect](#)

# Class Effect

Namespace: [Bliss.CSharp.Effects](#)

Assembly: Bliss.dll

```
public class Effect : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← Effect

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Constructors

## Effect(GraphicsDevice, VertexLayoutDescription, byte[], byte[], SpecializationConstant[]?)

Initializes a new instance of the [Effect](#) class using shader bytecode.

```
public Effect(GraphicsDevice graphicsDevice, VertexLayoutDescription vertexLayout, byte[] vertBytes, byte[] fragBytes, SpecializationConstant[]? constants = null)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The [GraphicsDevice](#) used for rendering.

**vertexLayout** [VertexLayoutDescription](#)

The [VertexLayoutDescription](#) defining the vertex structure.

**vertBytes** [byte](#)[]

The compiled bytecode for the vertex shader.

**fragBytes** [byte\[\]](#)

The compiled bytecode for the fragment shader.

**constants** [SpecializationConstant\[\]](#)

Optional specialization constants for shader customization.

**Effect(GraphicsDevice, VertexLayoutDescription, string, string, SpecializationConstant[]?)**

Initializes a new instance of the [Effect](#) class using shader file paths.

```
public Effect(GraphicsDevice graphicsDevice, VertexLayoutDescription vertexLayout, string vertPath, string fragPath, SpecializationConstant[]? constants = null)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The [GraphicsDevice](#) used for rendering.

**vertexLayout** [VertexLayoutDescription](#)

The [VertexLayoutDescription](#) defining the vertex structure.

**vertPath** [string](#)

The path to the vertex shader file.

**fragPath** [string](#)

The path to the fragment shader file.

**constants** [SpecializationConstant\[\]](#)

Optional specialization constants for shader customization.

## Fields

## Shader

Represents a pair of shaders consisting of a vertex shader and a fragment shader.

```
public readonly (Shader VertShader, Shader FragShader) Shader
```

### Field Value

[Shader](#) [VertShader](#), [Shader](#) [FragShader](#))

## ShaderSet

Represents a description of a shader set, including vertex layout details, shader information, and optional specialization constants.

```
public readonly ShaderSetDescription ShaderSet
```

### Field Value

[ShaderSetDescription](#)

## VertexLayout

Describes the layout of vertex data for a graphics pipeline.

```
public readonly VertexLayoutDescription VertexLayout
```

### Field Value

[VertexLayoutDescription](#)

## Properties

### GraphicsDevice

The graphics device used for creating and managing graphical resources.

```
public GraphicsDevice GraphicsDevice { get; }
```

## Property Value

[GraphicsDevice](#)

## Methods

### AddBufferLayout(SimpleBufferLayout)

Adds a buffer layout to the current effect instance.

```
public void AddBufferLayout(SimpleBufferLayout bufferLayout)
```

#### Parameters

**bufferLayout** [SimpleBufferLayout](#)

The buffer layout to be added, containing the name and configuration of the buffer.

### AddTextureLayout(SimpleTextureLayout)

Adds a texture layout to the effect.

```
public void AddTextureLayout(SimpleTextureLayout textureLayout)
```

#### Parameters

**textureLayout** [SimpleTextureLayout](#)

The texture layout to be added.

### Apply(Material?)

Apply the state effect immediately before rendering it.

```
public virtual void Apply(Material? material = null)
```

Parameters

`material` [Material](#)

## Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

Parameters

`disposing` [bool](#)

True if called from user code; false if called from a finalizer.

## GetBufferLayout(string)

Retrieves a buffer layout associated with the specified name.

```
public SimpleBufferLayout GetBufferLayout(string name)
```

Parameters

`name` [string](#)

The name of the buffer layout to retrieve.

Returns

[SimpleBufferLayout](#)

The [SimpleBufferLayout](#) associated with the given name.

## GetBufferLayoutKeys()

Retrieves all the keys from the buffer layout dictionary.

```
public string[] GetBufferLayoutKeys()
```

Returns

[string](#)[]

An array of strings representing the keys of the buffer layouts.

## GetBufferLayouts()

Retrieves an array of buffer layouts associated with the current effect.

```
public SimpleBufferLayout[] GetBufferLayouts()
```

Returns

[SimpleBufferLayout](#)[]

An array of [SimpleBufferLayout](#) objects, representing the buffer layouts used by the effect.

## GetPipeline(SimplePipelineDescription)

Retrieves or creates a pipeline for the given pipeline description.

```
public SimplePipeline GetPipeline(SimplePipelineDescription pipelineDescription)
```

Parameters

`pipelineDescription` [SimplePipelineDescription](#)

The description of the pipeline to retrieve or create.

Returns

## [SimplePipeline](#)

A [SimplePipeline](#) configured with the specified description.

### GetTextureLayout(string)

Retrieves a specific texture layout by its name.

```
public SimpleTextureLayout GetTextureLayout(string name)
```

Parameters

`name` [string](#)

The name of the texture layout to retrieve.

Returns

[SimpleTextureLayout](#)

The texture layout associated with the specified name.

### GetTextureLayoutKeys()

Retrieves an array of keys representing the names of texture layouts stored within the effect.

```
public string[] GetTextureLayoutKeys()
```

Returns

[string](#)[]

An array of strings containing the keys for the texture layouts.

### GetTextureLayouts()

Retrieves all texture layouts associated with the effect.

```
public SimpleTextureLayout[] GetTextureLayouts()
```

Returns

[SimpleTextureLayout\[\]](#)

An array of [SimpleTextureLayout](#) representing the texture layouts.

# Namespace Bliss.CSharp.Fonts

## Classes

[Font](#)

# Class Font

Namespace: [Bliss.CSharp.Fonts](#)

Assembly: Bliss.dll

```
public class Font : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← Font

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Constructors

## Font(byte[])

Initializes a new instance of the [Font](#) class with the specified font data.

```
public Font(byte[] data)
```

### Parameters

**data** [byte](#)[]

A byte array containing the font data.

## Font(string)

Initializes a new instance of the [Font](#) class, loading font data from the specified file path.

```
public Font(string path)
```

## Parameters

path [string](#)

The file path to the font data.

# Properties

## FontData

Gets the byte array containing the raw font data.

```
public byte[] FontData { get; }
```

## Property Value

[byte](#)[]

# Methods

## Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

## Parameters

disposing [bool](#)

True if called from user code; false if called from a finalizer.

Draw(SpriteBatch, string, Vector2, int, float, float, Vector2?,

# Vector2?, float, Color?, TextStyle, FontSystemEffect, int)

Draws text using the specified parameters.

```
public void Draw(SpriteBatch batch, string text, Vector2 position, int size, float characterSpacing = 0, float lineSpacing = 0, Vector2? scale = null, Vector2? origin = null, float rotation = 0, Color? color = null, TextStyle style = TextStyle.None, FontSystemEffect effect = FontSystemEffect.None, int effectAmount = 0)
```

## Parameters

**batch** [SpriteBatch](#)

The sprite batch used for rendering.

**text** [string](#)

The text to be drawn.

**position** [Vector2](#)

The position where the text should be drawn.

**size** [int](#)

The size of the font.

**characterSpacing** [float](#)

The spacing between characters. Default is 0.0F.

**lineSpacing** [float](#)

The spacing between lines. Default is 0.0F.

**scale** [Vector2](#)?

The scaling factor for the text. Default is null.

**origin** [Vector2](#)?

The origin for rotation and scaling. Default is null.

**rotation** [float](#)

The rotation angle for the text. Default is 0.0F.

**color** [Color](#)?

The color of the text. Default is white.

**style** TextStyle

The style of the text. Default is none.

**effect** FontSystemEffect

The effect to apply to the text. Default is none.

**effectAmount** [int](#)

The intensity of the applied effect. Default is 0.

## MeasureText(string, int)

Measures the dimensions of the specified text using the given font size.

```
public Vector2 MeasureText(string text, int size)
```

### Parameters

**text** [string](#)

The text to measure.

**size** [int](#)

The font size to use for measurement.

### Returns

[Vector2](#)

A Vector2 representing the width and height of the text.

## MeasureTextRect(string, int)

Measures the rectangular bounds of the given text with specified font size.

```
public RectangleF MeasureTextRect(string text, int size)
```

## Parameters

**text** [string](#)

The text to measure.

**size** [int](#)

The size of the font.

## Returns

[RectangleF](#)

## MeasureTextTrimmed(string, int)

Measures the dimensions of the specified trimmed text using the given font size.

```
public Vector2 MeasureTextTrimmed(string text, int size)
```

## Parameters

**text** [string](#)

The text to measure.

**size** [int](#)

The size of the font.

## Returns

[Vector2](#)

A [Vector2](#) representing the width and height of the trimmed text.

# Namespace Bliss.CSharp.Geometry

## Classes

[Mesh](#)

[Model](#)

## Structs

[BoundingBox](#)

# Struct BoundingBox

Namespace: [Bliss.CSharp.Geometry](#)

Assembly: Bliss.dll

```
public struct BoundingBox
```

## Inherited Members

[ValueType.Equals\(object\)](#) , [ValueType.GetHashCode\(\)](#) , [ValueType.ToString\(\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### BoundingBox(Vector3, Vector3)

Initializes a new instance of the [BoundingBox](#) struct with the specified minimum and maximum vectors.

```
public BoundingBox(Vector3 min, Vector3 max)
```

#### Parameters

**min** [Vector3](#)

The minimum vector defining one corner of the bounding box.

**max** [Vector3](#)

The maximum vector defining the opposite corner of the bounding box.

## Fields

### Max

Maximum box-corner.

```
public Vector3 Max
```

## Field Value

[Vector3](#) ↗

## Min

Minimum box-corner.

```
public Vector3 Min
```

## Field Value

[Vector3](#) ↗

# Class Mesh

Namespace: [Bliss.CSharp.Geometry](#)

Assembly: Bliss.dll

```
public class Mesh : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← Mesh

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Constructors

Mesh(GraphicsDevice, Material, Vertex3D[], uint[]?, Dictionary<string, Dictionary<int, BoneInfo[]>>?)

Initializes a new instance of the [Mesh](#) class with the specified properties.

```
public Mesh(GraphicsDevice graphicsDevice, Material material, Vertex3D[] vertices, uint[]?  
indices = null, Dictionary<string, Dictionary<int, BoneInfo[]>>? boneInfos = null)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device used to create buffers.

**material** [Material](#)

The material applied to the mesh.

**vertices** [Vertex3D\[\]](#)

The vertex data for the mesh.

### [indices](#) `uint[]`

The index data defining triangle order (optional).

### [boneInfos](#) `Dictionary<string, Dictionary<int, BoneInfo[]>>`

The skeletal bone mapping information for animations (optional).

## Properties

### BoneInfos

An array containing information about each bone in a mesh, used for skeletal animation. Each element provides details such as the bone's name, its identifier, and its transformation matrix.

```
public Dictionary<string, Dictionary<int, BoneInfo[]>> BoneInfos { get; }
```

### Property Value

`Dictionary<string, Dictionary<int, BoneInfo[]>>`

### BoundingBox

The axis-aligned bounding box (AABB) for the mesh. This bounding box is calculated based on the vertices of the mesh and represents the minimum and maximum coordinates that encompass the entire mesh.

```
public BoundingBox BoundingBox { get; }
```

### Property Value

`BoundingBox`

### GraphicsDevice

Represents the graphics device used for rendering operations. This property provides access to the underlying GraphicsDevice instance responsible for managing GPU resources and executing rendering commands.

```
public GraphicsDevice GraphicsDevice { get; }
```

## Property Value

[GraphicsDevice](#)

## IndexCount

The number of indices in the mesh used for rendering.

```
public uint IndexCount { get; }
```

## Property Value

[uint](#)

## Indices

An array of indices that define the order in which vertices are drawn. Indices are used in conjunction with the vertex array to form geometric shapes such as triangles in a mesh. This allows for efficient reuse of vertex data.

```
public uint[] Indices { get; }
```

## Property Value

[uint](#)[]

## Material

Represents the material properties used for rendering the mesh. This may include shaders (effects), texture mappings, blending states, and other rendering parameters. The Material controls how the mesh

is rendered within the graphics pipeline.

```
public Material Material { get; }
```

## Property Value

[Material](#)

## VertexCount

The total count of vertices present in the mesh. This value determines the number of vertices available for rendering within the mesh.

```
public uint VertexCount { get; }
```

## Property Value

[uint](#)

## Vertices

An array of Vertex3D structures that define the geometric points of a mesh. Each vertex contains attributes such as position, texture coordinates, normal, and optional color. Vertices are used to construct the shape and appearance of a 3D model.

```
public Vertex3D[] Vertices { get; }
```

## Property Value

[Vertex3D\[\]](#)

## Methods

### Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

## Parameters

**disposing** [bool](#)

True if called from user code; false if called from a finalizer.

## Draw(CommandList, Transform, OutputDescription, Sampler?, DepthStencilStateDescription?, RasterizerStateDescription?, Color?)

Renders the mesh with the specified properties and configurations.

```
public void Draw(CommandList commandList, Transform transform, OutputDescription output,
Sampler? sampler = null, DepthStencilStateDescription? depthStencilState = null,
RasterizerStateDescription? rasterizerState = null, Color? color = null)
```

## Parameters

**commandList** [CommandList](#)

The command list used to issue rendering commands.

**transform** [Transform](#)

The transformation applied to the mesh.

**output** [OutputDescription](#)

The output description specifying the rendering target and format.

**sampler** [Sampler](#)

The optional sampler state for texture sampling.

**depthStencilState** [DepthStencilStateDescription](#)?

The optional depth-stencil state description for depth testing.

**rasterizerState** [RasterizerStateDescription](#)?

The optional rasterizer state description for culling and rasterization.

### color [Color](#)?

An optional color to override the material's albedo map color.

## GenCapsule(GraphicsDevice, float, float, int)

Generates a capsule mesh with the specified radius, height, and number of slices.

```
public static Mesh GenCapsule(GraphicsDevice graphicsDevice, float radius, float height, int slices)
```

### Parameters

#### graphicsDevice [GraphicsDevice](#) ↗

The graphics device used to create and manage the mesh's GPU resources.

#### radius [float](#) ↗

The radius of the capsule.

#### height [float](#) ↗

The cylindrical midsection's height of the capsule.

#### slices [int](#) ↗

The number of slices used to approximate the capsule. Must be at least 3.

### Returns

#### [Mesh](#)

A [Mesh](#) representing a capsule with the given parameters.

## GenCone(GraphicsDevice, float, float, int)

Generates a 3D cone mesh with a specified radius, height, and number of slices.

```
public static Mesh GenCone(GraphicsDevice graphicsDevice, float radius, float height,  
int slices)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The [GraphicsDevice](#) used to manage GPU resources for the mesh.

**radius** [float](#)

The radius of the cone's base.

**height** [float](#)

The height of the cone from base to tip.

**slices** [int](#)

The number of slices dividing the circular base. Must be at least 3.

## Returns

[Mesh](#)

A new instance of the [Mesh](#) class representing the generated cone.

## GenCube(GraphicsDevice, float, float, float)

Generates a cuboid mesh with the specified dimensions and initializes it with default material and texture settings.

```
public static Mesh GenCube(GraphicsDevice graphicsDevice, float width, float height,  
float length)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device that manages GPU resources for the mesh.

**width** [float](#)

The width of the cuboid.

#### height [float](#)

The height of the cuboid.

#### length [float](#)

The length of the cuboid.

Returns

#### [Mesh](#)

A new instance of the [Mesh](#) class representing the cuboid.

## GenCylinder(GraphicsDevice, float, float, int)

Generates a cylindrical mesh with specified dimensions and resolution.

```
public static Mesh GenCylinder(GraphicsDevice graphicsDevice, float radius, float height, int slices)
```

Parameters

#### graphicsDevice [GraphicsDevice](#)

The [GraphicsDevice](#) responsible for managing GPU resources.

#### radius [float](#)

The radius of the cylinder's base and top.

#### height [float](#)

The height of the cylinder.

#### slices [int](#)

The number of slices (or segments) used to approximate the cylinder. Must be at least 3.

Returns

## [Mesh](#)

A new instance of [Mesh](#) representing the generated cylindrical geometry.

## GenHeightmap(GraphicsDevice, Image, Vector3)

Generates a heightmap-based 3D mesh from the given heightmap image and dimensions.

```
public static Mesh GenHeightmap(GraphicsDevice graphicsDevice, Image heightmap,  
Vector3 size)
```

### Parameters

**graphicsDevice** [GraphicsDevice](#)

The [GraphicsDevice](#) used for managing GPU resources and rendering the generated mesh.

**heightmap** [Image](#)

The heightmap image used to determine the height of each vertex in the mesh.

**size** [Vector3](#)

The 3D size of the heightmap mesh in world units, where X and Z represent the width and depth, and Y represents the height scale.

### Returns

[Mesh](#)

A new [Mesh](#) instance representing the generated heightmap mesh based on the input parameters.

## GenHemisphere(GraphicsDevice, float, int, int)

Generates a 3D hemisphere mesh with a specified radius, number of rings, and slices.

```
public static Mesh GenHemisphere(GraphicsDevice graphicsDevice, float radius, int rings,  
int slices)
```

### Parameters

## graphicsDevice [GraphicsDevice](#)

The [GraphicsDevice](#) instance used to allocate GPU resources for the mesh.

## radius [float](#)

The radius of the hemisphere.

## rings [int](#)

The number of subdivisions along the vertical (Y-axis) direction of the hemisphere.

## slices [int](#)

The number of subdivisions around the horizontal (XZ-plane) direction of the hemisphere.

## Returns

### [Mesh](#)

A new instance of the [Mesh](#) class representing the generated hemisphere model.

## GenKnot(GraphicsDevice, float, float, int, int)

Generates a torus knot mesh with the specified parameters.

```
public static Mesh GenKnot(GraphicsDevice graphicsDevice, float radius, float tubeRadius,
    int radSeg, int sides)
```

## Parameters

### graphicsDevice [GraphicsDevice](#)

The [GraphicsDevice](#) used to manage GPU resources for the mesh.

### radius [float](#)

The radius of the torus knot.

### tubeRadius [float](#)

The thickness of the tube forming the torus knot.

### radSeg [int](#)

The number of radial segments for the torus knot. The minimum value is 3.

#### sides [int](#)

The number of sides forming the cross-section of the tube. The minimum value is 3.

Returns

#### [Mesh](#)

A [Mesh](#) representing the torus knot with the specified parameters.

## GenPoly(GraphicsDevice, int, float)

Generates a 3D polygon mesh with the specified number of sides and radius.

```
public static Mesh GenPoly(GraphicsDevice graphicsDevice, int sides, float radius)
```

Parameters

#### graphicsDevice [GraphicsDevice](#)

The graphics device used to manage GPU resources for the generated mesh.

#### sides [int](#)

The number of sides of the polygon. Must be at least 3.

#### radius [float](#)

The radius of the polygon.

Returns

#### [Mesh](#)

A new instance of the [Mesh](#) class representing the 3D polygon.

## GenSphere(GraphicsDevice, float, int, int)

Generates a sphere mesh with the specified radius, rings, and slices.

```
public static Mesh GenSphere(GraphicsDevice graphicsDevice, float radius, int rings, int slices)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device used to create and manage the GPU resources needed for the mesh.

**radius** [float](#)

The radius of the sphere.

**rings** [int](#)

The number of horizontal segments dividing the sphere.

**slices** [int](#)

The number of vertical segments dividing the sphere.

## Returns

[Mesh](#)

A new instance of the [Mesh](#) class representing the generated sphere.

## GenTorus(GraphicsDevice, float, float, int, int)

Generates a torus-shaped mesh with specified dimensions and detail.

```
public static Mesh GenTorus(GraphicsDevice graphicsDevice, float radius, float size, int radSeg, int sides)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The [GraphicsDevice](#) used for managing GPU resources.

**radius** [float](#)

The radius of the torus from the center to the middle of the tube.

**size** [float](#)

The thickness of the torus tube.

**radSeg** [int](#)

The number of radial segments in the torus.

**sides** [int](#)

The number of segments around the tube's circular cross-section.

Returns

[Mesh](#)

A new instance of [Mesh](#) representing the generated torus.

## GenerateTangents()

Generates tangent vectors for the mesh's vertices based on the provided geometric and UV coordinate data.

```
public void GenerateTangents()
```

## ResetAnimationBones(CommandList)

Resets the bone transformation matrices to their identity state and updates the buffer on the GPU using the provided command list.

```
public void ResetAnimationBones(CommandList commandList)
```

Parameters

**commandList** [CommandList](#)

The command list used to record the buffer update command, ensuring the changes are applied to the GPU.

## UpdateAnimationBones(CommandList, ModelAnimation, int)

Updates the transformation matrices of the animation bones for a specific frame using the provided command list and animation data.

```
public void UpdateAnimationBones(CommandList commandList, ModelAnimation animation,  
int frame)
```

### Parameters

**commandList** [CommandList](#)

The command list used to issue rendering commands.

**animation** [ModelAnimation](#)

The animation data containing the bone transformations.

**frame** [int](#)

The specific frame of the animation to update the bone transformations.

# Class Model

Namespace: [Bliss.CSharp.Geometry](#)

Assembly: Bliss.dll

```
public class Model : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← Model

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Constructors

## Model(GraphicsDevice, Mesh[], ModelAnimation[])

Initializes a new instance of the [Model](#) class with the specified graphics device, meshes, and animations.

```
public Model(GraphicsDevice graphicsDevice, Mesh[] meshes, ModelAnimation[] animations)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The [GraphicsDevice](#) used for rendering and resource management.

**meshes** [Mesh](#)[]

An array of [Mesh](#) objects representing the geometric components of the model.

**animations** [ModelAnimation](#)[]

An array of [ModelAnimation](#) objects defining the animations for the model.

# Properties

## Animations

Collection of animations associated with the model. Each animation encapsulates skeletal transformations over time, enabling the model to exhibit complex motions.

```
public ModelAnimation[] Animations { get; }
```

### Property Value

[ModelAnimation\[\]](#)

## BoundingBox

Represents the axis-aligned bounding box of the model.

```
public BoundingBox BoundingBox { get; }
```

### Property Value

[BoundingBox](#)

## GraphicsDevice

The graphics device used for rendering the model.

```
public GraphicsDevice GraphicsDevice { get; }
```

### Property Value

[GraphicsDevice](#) ↗

## Meshes

An array of meshes that make up the model.

```
public Mesh[] Meshes { get; }
```

## Property Value

[Mesh\[\]](#)

## Methods

### Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

#### Parameters

`disposing` [bool](#)

True if called from user code; false if called from a finalizer.

### Draw(CommandList, Transform, OutputDescription, Sampler?, DepthStencilStateDescription?, RasterizerStateDescription?, Color?)

Draws the model using the specified command list, transform, and rendering configurations.

```
public void Draw(CommandList commandList, Transform transform, OutputDescription output,
Sampler? sampler = null, DepthStencilStateDescription? depthStencilState = null,
RasterizerStateDescription? rasterizerState = null, Color? color = null)
```

#### Parameters

`commandList` [CommandList](#)

The [CommandList](#) that issues the rendering commands.

`transform` [Transform](#)

The [Transform](#) describing the position, rotation, and scale of the model in the scene.

#### output [OutputDescription](#)

The [OutputDescription](#) that defines the target rendering output settings, such as resolution and format.

#### sampler [Sampler](#)

An optional [Sampler](#) used for texture sampling. If not provided, a default sampler is applied.

#### depthStencilState [DepthStencilStateDescription](#)?

An optional [DepthStencilStateDescription](#) to configure depth and stencil testing for the drawing process.

#### rasterizerState [RasterizerStateDescription](#)?

An optional [RasterizerStateDescription](#) that manages the rasterizer configuration, such as culling and fill mode.

#### color [Color](#)?

An optional [Color](#) to override or apply a custom tint to the model's rendered output.

## Load(GraphicsDevice, string, bool, bool)

Loads a model from the specified file path using the provided graphics device.

```
public static Model Load(GraphicsDevice graphicsDevice, string path, bool loadMaterial = true, bool flipUV = false)
```

### Parameters

#### graphicsDevice [GraphicsDevice](#)

The graphics device used for rendering the model.

#### path [string](#)

The file path to the model to be loaded.

#### loadMaterial [bool](#)

Indicates whether the material should be loaded with the model. Default is true.

#### **flipUv** [bool](#)

Indicates whether the UV coordinates should be flipped. Default is false.

Returns

#### [Model](#)

Returns a new instance of the [Model](#) class with the loaded meshes.

## ResetAnimationBones(CommandList)

Resets the animation bone transformations for all meshes in the model using the given command list.

```
public void ResetAnimationBones(CommandList commandList)
```

Parameters

#### **commandList** [CommandList](#)

The command list used to execute the reset operation on the GPU.

## UpdateAnimationBones(CommandList, ModelAnimation, int)

Updates the animation bones for all meshes in the model based on the specified animation and frame.

```
public void UpdateAnimationBones(CommandList commandList, ModelAnimation animation,  
int frame)
```

Parameters

#### **commandList** [CommandList](#)

The command list used to issue rendering commands.

#### **animation** [ModelAnimation](#)

The animation whose bone transformations should be applied.

## frame int

The specific frame of the animation to update the bones to.

# Namespace Bliss.CSharp.Geometry.Animations

## Classes

[BoneInfo](#)

[MeshAmateurBuilder](#)

[ModelAnimation](#)

[NodeAnimChannel](#)

# Class BoneInfo

Namespace: [Bliss.CSharp.Geometry.Animations](#)

Assembly: Bliss.dll

```
public class BoneInfo
```

## Inheritance

[object](#) ← BoneInfo

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### BoneInfo(string, uint, Matrix4x4)

Initializes a new instance of the [BoneInfo](#) class with the specified name, ID, and transformation.

```
public BoneInfo(string name, uint id, Matrix4x4 transformation)
```

## Parameters

**name** [string](#)

The name of the bone.

**id** [uint](#)

The unique identifier of the bone.

**transformation** [Matrix4x4](#)

The transformation matrix for the bone.

## Properties

## Id

The unique identifier of the bone.

```
public uint Id { get; }
```

## Property Value

[uint](#)

## Name

The name of the bone.

```
public string Name { get; }
```

## Property Value

[string](#)

## Transformation

The transformation matrix representing the bone's transformation in the skeleton hierarchy.

```
public Matrix4x4 Transformation { get; }
```

## Property Value

[Matrix4x4](#)

# Class MeshAmateurBuilder

Namespace: [Bliss.CSharp.Geometry.Animations](#)

Assembly: Bliss.dll

```
public class MeshAmateurBuilder
```

## Inheritance

[object](#) ← MeshAmateurBuilder

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### MeshAmateurBuilder(Node, ModelAnimation[])

Initializes a new instance of the [MeshAmateurBuilder](#) class with a root node and animations.

```
public MeshAmateurBuilder(Node rootNode, ModelAnimation[] animations)
```

#### Parameters

**rootNode** Node

The root Assimp.Node representing the hierarchical structure of the mesh.

**animations** [ModelAnimation](#)[]

An array of [ModelAnimation](#) objects defining animations for the mesh.

## Methods

### Build(Dictionary<uint, Bone>)

Constructs a dictionary containing bone information mapped by animation name and frame index.

```
public Dictionary<string, Dictionary<int, BoneInfo[]>> Build(Dictionary<uint, Bone> bonesByName)
```

## Parameters

bonesByName [Dictionary](#)<[uint](#), Bone>

A dictionary mapping bone identifiers to bone objects.

## Returns

[Dictionary](#)<[string](#), [Dictionary](#)<[int](#), [BoneInfo](#)[]>>

A dictionary where keys are animation names, and values are dictionaries that map frame indices to arrays of bone information.

# Class ModelAnimation

Namespace: [Bliss.CSharp.Geometry.Animations](#)

Assembly: Bliss.dll

```
public class ModelAnimation
```

## Inheritance

[object](#) ← ModelAnimation

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

**ModelAnimation(string, float, float,  
IReadOnlyList<NodeAnimChannel>)**

Initializes a new instance of the [ModelAnimation](#) class with the specified properties and animation channels.

```
public ModelAnimation(string name, float durationInTicks, float ticksPerSecond,  
IReadOnlyList<NodeAnimChannel> channels)
```

## Parameters

**name** [string](#)

The name of the animation.

**durationInTicks** [float](#)

The total duration of the animation in ticks.

**ticksPerSecond** [float](#)

The number of ticks per second, determining playback speed.

`channels` [IReadOnlyList](#)<[NodeAnimChannel](#)>

The list of animation channels associated with this animation.

## Properties

### AnimationChannels

A list of animation channels that define the per-node transformations during the animation.

```
public IReadOnlyList<NodeAnimChannel> AnimationChannels { get; }
```

Property Value

[IReadOnlyList](#)<[NodeAnimChannel](#)>

### DurationInTicks

The total duration of the animation in ticks.

```
public float DurationInTicks { get; }
```

Property Value

[float](#)

### FrameCount

The total number of frames in the animation, calculated based on duration and ticks per second.

```
public int FrameCount { get; }
```

Property Value

[int](#)

## Name

The name of the animation.

```
public string Name { get; }
```

## Property Value

[string](#)

## TicksPerSecond

The number of ticks per second for the animation, determining its playback speed.

```
public float TicksPerSecond { get; }
```

## Property Value

[float](#)

# Class NodeAnimChannel

Namespace: [Bliss.CSharp.Geometry.Animations](#)

Assembly: Bliss.dll

```
public class NodeAnimChannel
```

## Inheritance

[object](#) ← NodeAnimChannel

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

**NodeAnimChannel(string, IReadOnlyList<Vector3Key>, IReadOnlyList<QuatKey>, IReadOnlyList<Vector3Key>)**

Initializes a new instance of the [NodeAnimChannel](#) class with the specified name, positions, rotations, and scales.

```
public NodeAnimChannel(string name, IReadOnlyList<Vector3Key> positions,  
IReadOnlyList<QuatKey> rotations, IReadOnlyList<Vector3Key> scales)
```

## Parameters

**name** [string](#)

The name of the node being animated.

**positions** [IReadOnlyList](#)<[Vector3Key](#)>

A list of [Vector3Key](#) representing position keyframes.

**rotations** [IReadOnlyList](#)<[QuatKey](#)>

A list of [QuatKey](#) representing rotation keyframes.

scales [IReadOnlyList](#)<[Vector3Key](#)>

A list of [Vector3Key](#) representing scale keyframes.

## Properties

### Name

The name of the node being animated.

```
public string Name { get; }
```

### Property Value

[string](#)

### Positions

A list of [Vector3Key](#) representing the position keyframes for the node.

```
public IReadOnlyList<Vector3Key> Positions { get; }
```

### Property Value

[IReadOnlyList](#)<[Vector3Key](#)>

### Rotations

A list of [QuatKey](#) representing the rotation keyframes (quaternions) for the node.

```
public IReadOnlyList<QuatKey> Rotations { get; }
```

### Property Value

[IReadOnlyList](#)<[QuatKey](#)>

## Scales

A list of [Vector3Key](#) representing the scale keyframes for the node.

```
public IReadOnlyList<Vector3Key> Scales { get; }
```

Property Value

[IReadOnlyList](#) <[Vector3Key](#)>

# Namespace Bliss.CSharp.Geometry.Animations.Keyframes

## Structs

[QuatKey](#)

[Vector3Key](#)

# Struct QuatKey

Namespace: [Bliss.CSharp.Geometry.Animations.Keyframes](#)

Assembly: Bliss.dll

```
public struct QuatKey : IEquatable<QuatKey>
```

Implements

[IEquatable](#)<[QuatKey](#)>

Inherited Members

[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### QuatKey(double, Quaternion)

Initializes a new instance of the [QuatKey](#) struct with the specified time and value.

```
public QuatKey(double time, Quaternion value)
```

Parameters

**time** [double](#)

The time of the keyframe.

**value** [Quaternion](#)

The quaternion value (e.g., rotation) at the specified time.

## Fields

### Time

The time at which this keyframe occurs.

```
public double Time
```

Field Value

[double](#)

Value

The quaternion value (e.g., rotation) at the specified time.

```
public Quaternion Value
```

Field Value

[Quaternion](#)

## Methods

### Equals(QuatKey)

Compares this [QuatKey](#) to another for equality based on their quaternion value.

```
public bool Equals(QuatKey other)
```

Parameters

**other** [QuatKey](#)

The other [QuatKey](#) to compare.

Returns

[bool](#)

Returns true if the quaternion values of both keys are equal, otherwise false.

## Equals(object?)

Compares this [QuatKey](#) to another object for equality.

```
public override bool Equals(object? obj)
```

Parameters

**obj** [object](#)

The object to compare to.

Returns

[bool](#)

Returns true if the object is a [QuatKey](#) and has the same quaternion value.

## GetHashCode()

Generates a hash code based on the quaternion value of the [QuatKey](#).

```
public override int GetHashCode()
```

Returns

[int](#)

The hash code for this key.

## ToString()

Returns a string representation of the [QuatKey](#) including its time and quaternion value.

```
public override string ToString()
```

Returns

[string](#)

A string representing the [QuatKey](#).

## Operators

### operator ==(QuatKey, QuatKey)

Compares two [QuatKey](#) instances for equality based on their quaternion value.

```
public static bool operator ==(QuatKey left, QuatKey right)
```

Parameters

**left** [QuatKey](#)

The first [QuatKey](#) to compare.

**right** [QuatKey](#)

The second [QuatKey](#) to compare.

Returns

[bool](#)

Returns true if the quaternion values of both keys are equal, otherwise false.

### operator >(QuatKey, QuatKey)

Compares two [QuatKey](#) instances to see if the first occurs after the second based on their time.

```
public static bool operator >(QuatKey left, QuatKey right)
```

Parameters

**left** [QuatKey](#)

The first [QuatKey](#) to compare.

**right** [QuatKey](#)

The second [QuatKey](#) to compare.

Returns

[bool](#) ↗

Returns true if the first key occurs after the second key in time.

## operator !=(QuatKey, QuatKey)

Compares two [QuatKey](#) instances for inequality based on their quaternion value.

```
public static bool operator !=(QuatKey left, QuatKey right)
```

Parameters

**left** [QuatKey](#)

The first [QuatKey](#) to compare.

**right** [QuatKey](#)

The second [QuatKey](#) to compare.

Returns

[bool](#) ↗

Returns true if the quaternion values of both keys are not equal, otherwise false.

## operator <(QuatKey, QuatKey)

Compares two [QuatKey](#) instances to see if the first occurs before the second based on their time.

```
public static bool operator <(QuatKey left, QuatKey right)
```

Parameters

**left** [QuatKey](#)

The first [QuatKey](#) to compare.

**right** [QuatKey](#)

The second [QuatKey](#) to compare.

Returns

[bool](#) ↗

Returns true if the first key occurs before the second key in time.

# Struct Vector3Key

Namespace: [Bliss.CSharp.Geometry.Animations.Keyframes](#)

Assembly: Bliss.dll

```
public struct Vector3Key : IEquatable<Vector3Key>
```

## Implements

[IEquatable](#) <[Vector3Key](#)>

## Inherited Members

[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### Vector3Key(double, Vector3)

Initializes a new instance of the [Vector3Key](#) struct with the specified time and value.

```
public Vector3Key(double time, Vector3 value)
```

#### Parameters

**time** [double](#)

The time of the keyframe.

**value** [Vector3](#)

The 3D vector value at the specified time.

## Fields

### Time

The time at which this keyframe occurs.

```
public double Time
```

Field Value

[double](#)

Value

The 3D vector value (e.g., position, rotation) at the specified time.

```
public Vector3 Value
```

Field Value

[Vector3](#)

## Methods

### Equals(Vector3Key)

Compares this [Vector3Key](#) to another for equality based on their value.

```
public bool Equals(Vector3Key other)
```

Parameters

other [Vector3Key](#)

The other [Vector3Key](#) to compare.

Returns

[bool](#)

Returns true if the value of both keys are equal, otherwise false.

## Equals(object?)

Compares this [Vector3Key](#) to another object for equality.

```
public override bool Equals(object? obj)
```

Parameters

**obj** [object](#)

The object to compare to.

Returns

[bool](#)

Returns true if the object is a [Vector3Key](#) and has the same value.

## GetHashCode()

Generates a hash code based on the value of the [Vector3Key](#).

```
public override int GetHashCode()
```

Returns

[int](#)

The hash code for this key.

## ToString()

Returns a string representation of the [Vector3Key](#) including its time and value.

```
public override string ToString()
```

Returns

[string](#)

A string representing the [Vector3Key](#).

## Operators

### operator ==(Vector3Key, Vector3Key)

Compares two [Vector3Key](#) instances for equality based on their value.

```
public static bool operator ==(Vector3Key left, Vector3Key right)
```

Parameters

**left** [Vector3Key](#)

The first [Vector3Key](#) to compare.

**right** [Vector3Key](#)

The second [Vector3Key](#) to compare.

Returns

[bool](#)

Returns true if the value of both keys are equal, otherwise false.

### operator >(Vector3Key, Vector3Key)

Compares two [Vector3Key](#) instances to see if the first occurs after the second based on their time.

```
public static bool operator >(Vector3Key left, Vector3Key right)
```

Parameters

**left** [Vector3Key](#)

The first [Vector3Key](#) to compare.

**right** [Vector3Key](#)

The second [Vector3Key](#) to compare.

Returns

[bool](#) ↗

Returns true if the first key occurs after the second key in time.

## operator !=(Vector3Key, Vector3Key)

Compares two [Vector3Key](#) instances for inequality based on their value.

```
public static bool operator !=(Vector3Key left, Vector3Key right)
```

Parameters

**left** [Vector3Key](#)

The first [Vector3Key](#) to compare.

**right** [Vector3Key](#)

The second [Vector3Key](#) to compare.

Returns

[bool](#) ↗

Returns true if the value of both keys are not equal, otherwise false.

## operator <(Vector3Key, Vector3Key)

Compares two [Vector3Key](#) instances to see if the first occurs before the second based on their time.

```
public static bool operator <(Vector3Key left, Vector3Key right)
```

Parameters

**left** [Vector3Key](#)

The first [Vector3Key](#) to compare.

**right** [Vector3Key](#)

The second [Vector3Key](#) to compare.

Returns

[bool](#) ↗

Returns true if the first key occurs before the second key in time.

# Namespace Bliss.CSharp.Graphics

## Classes

[GraphicsHelper](#)

## Enums

[SamplerType](#)

# Class GraphicsHelper

Namespace: [Bliss.CSharp.Graphics](#)

Assembly: Bliss.dll

```
public static class GraphicsHelper
```

## Inheritance

[object](#) ← GraphicsHelper

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Methods

### GetSampler(GraphicsDevice, SamplerType)

Retrieves a Sampler object based on the provided SamplerType.

```
public static Sampler GetSampler(GraphicsDevice graphicsDevice, SamplerType samplerType)
```

#### Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device used to create the sampler.

**samplerType** [SamplerType](#)

The type of sampler to retrieve.

#### Returns

[Sampler](#)

A Sampler object corresponding to the specified SamplerType.

#### Exceptions

## [ArgumentException](#)

Thrown when an unsupported sampler type is provided.

# Enum SamplerType

Namespace: [Bliss.CSharp.Graphics](#)

Assembly: Bliss.dll

```
public enum SamplerType
```

## Fields

**Aniso4X = 2**

Anisotropic sampling with a maximum of 4x sampling, providing better quality at glancing angles.

**Linear = 1**

Linear sampling, which performs linear interpolation between texels.

**Point = 0**

Point sampling, which selects the nearest texel without filtering.

# Namespace Bliss.CSharp.Graphics.Pipelines

## Classes

[SimplePipeline](#)

## Structs

[SimplePipelineDescription](#)

# Class SimplePipeline

Namespace: [Bliss.CSharp.Graphics.Pipelines](#)

Assembly: Bliss.dll

```
public class SimplePipeline : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← SimplePipeline

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### SimplePipeline(GraphicsDevice, SimplePipelineDescription)

Initializes a new instance of the [SimplePipeline](#) class using the provided graphics device and pipeline description.

```
public SimplePipeline(GraphicsDevice graphicsDevice, SimplePipelineDescription  
pipelineDescription)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device that will be used to create the pipeline.

**pipelineDescription** [SimplePipelineDescription](#)

The description of the pipeline, containing configurations like blend state, shader set, and resource layouts.

# Fields

## PipelineDescription

Contains the configuration details for setting up a graphics pipeline, including states and resource layouts.

```
public SimplePipelineDescription PipelineDescription
```

### Field Value

[SimplePipelineDescription](#)

# Properties

## GraphicsDevice

Represents the graphics device used by the pipeline for creating and managing graphics resources.

```
public GraphicsDevice GraphicsDevice { get; }
```

### Property Value

[GraphicsDevice](#)

## Pipeline

Represents the graphics pipeline used to render graphics objects.

```
public Pipeline Pipeline { get; }
```

### Property Value

[Pipeline](#)

## ResourceLayouts

Represents the array of resource layouts used by the pipeline to manage buffer and texture resources.

```
public ResourceLayout[] ResourceLayouts { get; }
```

Property Value

[ResourceLayout](#)[]

## Methods

### Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

Parameters

**disposing** [bool](#)

True if called from user code; false if called from a finalizer.

### GetBufferLayout(string)

Retrieves a buffer layout from the pipeline description by its name.

```
public SimpleBufferLayout? GetBufferLayout(string name)
```

Parameters

**name** [string](#)

The name of the buffer layout to retrieve.

Returns

[SimpleBufferLayout](#)

A [SimpleBufferLayout](#) object if a matching buffer layout is found; otherwise, null.

## GetTextureLayout(string)

Retrieves the [SimpleTextureLayout](#) with the specified name from the pipeline description's texture layouts.

```
public SimpleTextureLayout? GetTextureLayout(string name)
```

### Parameters

**name** [string](#) ↗

The name of the texture layout to retrieve.

### Returns

[SimpleTextureLayout](#)

The [SimpleTextureLayout](#) if a matching layout is found; otherwise, null.

# Struct SimplePipelineDescription

Namespace: [Bliss.CSharp.Graphics.Pipelines](#)

Assembly: Bliss.dll

```
public struct SimplePipelineDescription : IEquatable<SimplePipelineDescription>
```

Implements

[IEquatable](#)<[SimplePipelineDescription](#)>

Inherited Members

[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### SimplePipelineDescription()

Initializes a new instance of the [SimplePipelineDescription](#) struct.

```
public SimplePipelineDescription()
```

### SimplePipelineDescription(BlendStateDescription, DepthStencilStateDescription, RasterizerStateDescription, PrimitiveTopology, SimpleBufferLayout[], SimpleTextureLayout[], ShaderSetDescription, OutputDescription)

Initializes a new instance of the [SimplePipelineDescription](#) struct with the provided pipeline configuration details.

```
public SimplePipelineDescription(BlendStateDescription blendState,  
DepthStencilStateDescription depthStencilState, RasterizerStateDescription  
rasterizerState, PrimitiveTopology primitiveTopology, SimpleBufferLayout[]  
bufferLayouts, SimpleTextureLayout[] textureLayouts, ShaderSetDescription shaderSet,  
OutputDescription outputs)
```

## Parameters

**blendState** [BlendStateDescription](#)

The blend state configuration of the pipeline.

**depthStencilState** [DepthStencilStateDescription](#)

The depth and stencil state configuration of the pipeline.

**rasterizerState** [RasterizerStateDescription](#)

The rasterizer state configuration of the pipeline.

**primitiveTopology** [PrimitiveTopology](#)

The primitive topology that defines how vertex data is interpreted.

**bufferLayouts** [SimpleBufferLayout](#)[]

An array of buffer layouts to be used by the pipeline.

**textureLayouts** [SimpleTextureLayout](#)[]

An array of texture layouts to be used in the pipeline.

**shaderSet** [ShaderSetDescription](#)

The shader set description that defines the vertex and fragment shaders.

**outputs** [OutputDescription](#)[]

The output configuration of the pipeline, specifying the render targets and depth-stencil buffer.

## Fields

### BlendState

Defines the blend state configuration, controlling how colors are blended in the pipeline.

```
public BlendStateDescription BlendState
```

Field Value

[BlendStateDescription](#)

## BufferLayouts

An array of buffers that are bound to the pipeline, containing vertex data and possibly other information.

```
public SimpleBufferLayout[] BufferLayouts
```

Field Value

[SimpleBufferLayout\[\]](#)

## DepthStencilState

Defines the depth and stencil state configuration, controlling depth testing and stencil operations in the pipeline.

```
public DepthStencilStateDescription DepthStencilState
```

Field Value

[DepthStencilStateDescription](#)

## Outputs

Defines the output configuration, specifying render targets and the depth-stencil buffer for the pipeline.

```
public OutputDescription Outputs
```

Field Value

[OutputDescription](#)

## PrimitiveTopology

Specifies the primitive topology, which defines how the input vertices are interpreted (e.g., triangles, lines).

```
public PrimitiveTopology PrimitiveTopology
```

Field Value

[PrimitiveTopology](#) ↗

## RasterizerState

Defines the rasterizer state configuration, which determines how primitives are rasterized (e.g., culling mode, fill mode).

```
public RasterizerStateDescription RasterizerState
```

Field Value

[RasterizerStateDescription](#) ↗

## ResourceBindingModel

An optional resource binding model that maps resources such as textures and buffers to the pipeline.

```
public ResourceBindingModel? ResourceBindingModel
```

Field Value

[ResourceBindingModel](#) ↗?

## ShaderSet

Describes the shader set, including the vertex and fragment shaders used by the pipeline.

```
public ShaderSetDescription ShaderSet
```

Field Value

[ShaderSetDescription](#) ↗

## TextureLayouts

An array of texture layouts, describing how textures are arranged and accessed in the pipeline.

```
public SimpleTextureLayout[] TextureLayouts
```

Field Value

[SimpleTextureLayout\[\]](#)

## Methods

### Equals(SimplePipelineDescription)

Determines whether the current [SimplePipelineDescription](#) instance is equal to another specified instance.

```
public bool Equals(SimplePipelineDescription other)
```

Parameters

**other** [SimplePipelineDescription](#)

The other [SimplePipelineDescription](#) instance to compare with the current instance.

Returns

[bool](#) ↗

A boolean value indicating whether the two instances are equal.

### Equals(object?)

Determines whether the specified object is equal to the current [SimplePipelineDescription](#) instance.

```
public override bool Equals(object? obj)
```

Parameters

**obj** [object](#)

The object to compare with the current instance.

Returns

[bool](#)

A boolean value indicating whether the specified object is equal to the current instance.

## GetHashCode()

Returns a hash code for this instance of [SimplePipelineDescription](#).

```
public override int GetHashCode()
```

Returns

[int](#)

A 32-bit signed integer hash code that is representative of the object's current state and its members.

## ToString()

Returns a string representation of the [SimplePipelineDescription](#) instance, detailing its configuration and properties.

```
public override string ToString()
```

Returns

[string](#)

A string that includes the blend state, depth-stencil state, rasterizer state, primitive topology, buffers, texture layouts, shader set, outputs, and resource binding model of the pipeline description.

## Operators

### operator ==(SimplePipelineDescription, SimplePipelineDescription)

Determines whether two [SimplePipelineDescription](#) instances are equal based on their properties.

```
public static bool operator ==(SimplePipelineDescription left, SimplePipelineDescription right)
```

#### Parameters

##### **left** [SimplePipelineDescription](#)

The first [SimplePipelineDescription](#) instance to compare.

##### **right** [SimplePipelineDescription](#)

The second [SimplePipelineDescription](#) instance to compare.

#### Returns

##### [bool](#) ↗

**true** if the instances are equal; otherwise, **false**.

### operator !=(SimplePipelineDescription, SimplePipelineDescription)

Determines whether two [SimplePipelineDescription](#) instances are equal.

```
public static bool operator !=(SimplePipelineDescription left, SimplePipelineDescription right)
```

#### Parameters

**left** [SimplePipelineDescription](#)

The first [SimplePipelineDescription](#) to compare.

**right** [SimplePipelineDescription](#)

The second [SimplePipelineDescription](#) to compare.

Returns

[bool](#) ↗

True if both instances are equal; otherwise, false.

# Namespace Bliss.CSharp.Graphics.Pipelines.Buffers

## Classes

[SimpleBufferLayout](#)

[SimpleBuffer<T>](#)

## Interfaces

[ISimpleBuffer](#)

## Enums

[SimpleBufferType](#)

# Interface ISimpleBuffer

Namespace: [Bliss.CSharp.Graphics.Pipelines.Buffers](#)

Assembly: Bliss.dll

```
public interface ISimpleBuffer : IDisposable
```

## Inherited Members

[IDisposable.Dispose\(\)](#) ↗

## Properties

### DeviceBuffer

The device buffer associated with this instance.

```
DeviceBuffer DeviceBuffer { get; }
```

### Property Value

[DeviceBuffer](#) ↗

# Class SimpleBufferLayout

Namespace: [Bliss.CSharp.Graphics.Pipelines.Buffers](#)

Assembly: Bliss.dll

```
public class SimpleBufferLayout : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← SimpleBufferLayout

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### SimpleBufferLayout(GraphicsDevice, string, SimpleBufferType, ShaderStages)

Creates a new instance of the [SimpleBufferLayout](#) class with the specified parameters.

```
public SimpleBufferLayout(GraphicsDevice graphicsDevice, string name, SimpleBufferType  
bufferType, ShaderStages stages)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device used for resource allocation.

**name** [string](#)

The name of the buffer layout.

**bufferType** [SimpleBufferType](#)

The type of buffer (e.g., Uniform, Structured).

### **stages** [ShaderStages](#)

The shader stages where this layout will be used.

## Properties

### BufferType

The type of the buffer, determining its usage and binding.

```
public SimpleBufferType BufferType { get; }
```

Property Value

[SimpleBufferType](#)

### GraphicsDevice

The graphics device used to create the layout and associated resources.

```
public GraphicsDevice GraphicsDevice { get; }
```

Property Value

[GraphicsDevice](#)

### Layout

The resource layout representing the buffer's structure in the graphics pipeline.

```
public ResourceLayout Layout { get; }
```

Property Value

[ResourceLayout](#)

## Name

The name assigned to the buffer layout.

```
public string Name { get; }
```

## Property Value

[string](#)

## ShaderStages

The shader stages where this buffer layout is accessible.

```
public ShaderStages ShaderStages { get; }
```

## Property Value

[ShaderStages](#)

## Methods

### Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

## Parameters

**disposing** [bool](#)

True if called from user code; false if called from a finalizer.

# Enum SimpleBufferType

Namespace: [Bliss.CSharp.Graphics.Pipelines.Buffers](#)

Assembly: Bliss.dll

```
public enum SimpleBufferType
```

## Fields

**StructuredReadOnly = 1**

A read-only buffer that supports structured data access.

**StructuredReadWrite = 2**

A read-write buffer that supports structured data access.

**Uniform = 0**

A buffer type that holds uniform data.

# Class SimpleBuffer<T>

Namespace: [Bliss.CSharp.Graphics.Pipelines.Buffers](#)

Assembly: Bliss.dll

```
public class SimpleBuffer<T> : Disposable, ISimpleBuffer, IDisposable where T : unmanaged
```

## Type Parameters

T

## Inheritance

[object](#) ← [Disposable](#) ← SimpleBuffer<T>

## Implements

[ISimpleBuffer](#), [IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#), [Disposable.Dispose\(\)](#), [Disposable.ThrowIfDisposed\(\)](#), [object.Equals\(object\)](#),  
[object.Equals\(object, object\)](#), [object.GetHashCode\(\)](#), [object.GetType\(\)](#),  
[object.MemberwiseClone\(\)](#), [object.ReferenceEquals\(object, object\)](#), [object.ToString\(\)](#)

## Constructors

### SimpleBuffer(GraphicsDevice, uint, SimpleBufferType, ShaderStages)

Initializes a new instance of the [SimpleBuffer<T>](#) class with the specified graphics device, buffer name, size, buffer type, and shader stages.

```
public SimpleBuffer(GraphicsDevice graphicsDevice, uint size, SimpleBufferType bufferType,  
ShaderStages stages)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device used to create the buffer and related resources.

**size** [uint](#)

The size of the buffer in elements.

**bufferType** [SimpleBufferType](#)

The type of the buffer, which defines its usage.

**stages** [ShaderStages](#)

The shader stages where this buffer will be used.

## Properties

### BufferType

The type of the buffer, which defines its usage.

```
public SimpleBufferType BufferType { get; }
```

Property Value

[SimpleBufferType](#)

### Data

An array containing the data stored in the buffer.

```
public T[] Data { get; }
```

Property Value

T[]

### DeviceBuffer

Represents the buffer resource allocated on the graphics device.

```
public DeviceBuffer DeviceBuffer { get; }
```

Property Value

[DeviceBuffer](#)

## GraphicsDevice

The graphics device used to create the buffer and related resources.

```
public GraphicsDevice GraphicsDevice { get; }
```

Property Value

[GraphicsDevice](#)

## ShaderStages

The shader stages where this buffer will be used.

```
public ShaderStages ShaderStages { get; }
```

Property Value

[ShaderStages](#)

## Size

The size of the buffer in elements.

```
public uint Size { get; }
```

Property Value

[uint](#)

# Methods

## Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

Parameters

`disposing` [bool](#)

True if called from user code; false if called from a finalizer.

## GetResourceSet(SimpleBufferLayout)

Retrieves a cached resource set or creates a new one based on the provided buffer layout.

```
public ResourceSet GetResourceSet(SimpleBufferLayout layout)
```

Parameters

`layout` [SimpleBufferLayout](#)

The layout defining the structure and configuration of the resource set.

Returns

[ResourceSet](#)

The resource set corresponding to the specified layout.

## SetValue(int, T)

Sets the value at the specified index in the buffer.

```
public void SetValue(int index, T value)
```

## Parameters

**index** [int](#)

The index in the buffer where the value should be set.

**value** [T](#)

The value to set in the buffer.

## SetValueDeferred(CommandList, int, T)

Sets the value at the specified index in the buffer and schedules a command to update the buffer on the command list.

```
public void SetValueDeferred(CommandList commandList, int index, T value)
```

## Parameters

**commandList** [CommandList](#)

The command list to which the buffer update command will be added.

**index** [int](#)

The index of the buffer element to set.

**value** [T](#)

The value to set at the specified index.

## SetValueImmediate(int, T)

Sets the value of the buffer at the specified index and updates the buffer on the graphics device immediately.

```
public void SetValueImmediate(int index, T value)
```

## Parameters

**index** [int](#)

The index at which the value should be set.

**value** T

The value to set at the specified index.

## UpdateBuffer(CommandList)

Updates the contents of the device buffer with the current data.

```
public void UpdateBuffer(CommandList commandList)
```

### Parameters

**commandList** [CommandList](#)

The command list used to record the buffer update command.

## UpdateBufferImmediate()

Immediately updates the GPU buffer with the current data held in the CPU buffer.

```
public void UpdateBufferImmediate()
```

# Namespace Bliss.CSharp.Graphics.Pipelines.Textures

## Classes

[SimpleTextureLayout](#)

# Class SimpleTextureLayout

Namespace: [Bliss.CSharp.Graphics.Pipelines.Textures](#)

Assembly: Bliss.dll

```
public class SimpleTextureLayout : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← SimpleTextureLayout

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### SimpleTextureLayout(GraphicsDevice, string)

Initializes a new instance of the [SimpleTextureLayout](#) class with the specified graphics device and texture name.

```
public SimpleTextureLayout(GraphicsDevice graphicsDevice, string name)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device used to create the resource layout.

**name** [string](#)

The name used for the texture and sampler resources.

# Properties

## GraphicsDevice

The graphics device used to create the resource layout for textures and samplers.

```
public GraphicsDevice GraphicsDevice { get; }
```

### Property Value

[GraphicsDevice](#) ↗

## Layout

The resource layout associated with textures and samplers.

```
public ResourceLayout Layout { get; }
```

### Property Value

[ResourceLayout](#) ↗

## Name

The name used for the texture and sampler resources.

```
public string Name { get; }
```

### Property Value

[string](#) ↗

# Methods

## Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

## Parameters

**disposing** [bool](#)

True if called from user code; false if called from a finalizer.

# Namespace Bliss.CSharp.Graphics.Rendering

## Classes

[Frustum](#)

# Class Frustum

Namespace: [Bliss.CSharp.Graphics.Rendering](#)

Assembly: Bliss.dll

```
public class Frustum
```

## Inheritance

[object](#) ← Frustum

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Constructors

## Frustum()

Initializes a new instance of the Frustum class.

```
public Frustum()
```

# Methods

## ContainsBox(BoundingBox)

Checks if a bounding box is contained within the frustum.

```
public bool ContainsBox(BoundingBox box)
```

## Parameters

box [BoundingBox](#)

The bounding box to check.

Returns

[bool](#)

True if the bounding box is contained within the frustum, otherwise false.

## ContainsOrientedBox(BoundingBox, Vector3, Quaternion)

Checks if a oriented box is contained within the frustum.

```
public bool ContainsOrientedBox(BoundingBox box, Vector3 origin, Quaternion rotation)
```

Parameters

[box](#) [BoundingBox](#)

The oriented box to check.

[origin](#) [Vector3](#)

The origin of the oriented box.

[rotation](#) [Quaternion](#)

The rotation of the oriented box.

Returns

[bool](#)

True if the oriented box is contained within the frustum, otherwise false.

## ContainsPoint(Vector3)

Checks if a point is contained within the frustum.

```
public bool ContainsPoint(Vector3 point)
```

Parameters

**point** [Vector3](#)

The point to check.

Returns

[bool](#)

True if the point is contained within the frustum, otherwise false.

## ContainsSphere(Vector3, float)

Checks if a sphere is contained within the frustum.

```
public bool ContainsSphere(Vector3 center, float radius)
```

Parameters

**center** [Vector3](#)

The center of the sphere.

**radius** [float](#)

The radius of the sphere.

Returns

[bool](#)

True if the sphere is contained within the frustum, otherwise false.

## Extract(Matrix4x4)

Extracts frustum planes from the view-projection matrix.

```
public void Extract(Matrix4x4 viewProjection)
```

Parameters

`viewProjection Matrix4x4`

# Namespace Bliss.CSharp.Graphics.Rendering. Batches.Primitives

## Classes

[PrimitiveBatch](#)

# Class PrimitiveBatch

Namespace: [Bliss.CSharp.Graphics.Rendering.Batches.Primitives](#)

Assembly: Bliss.dll

```
public class PrimitiveBatch : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← PrimitiveBatch

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Constructors

## PrimitiveBatch(GraphicsDevice, IWindow, uint)

Initializes a new instance of the PrimitiveBatch class for rendering 2D primitives.

```
public PrimitiveBatch(GraphicsDevice graphicsDevice, IWindow window, uint capacity = 30720)
```

### Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device used for rendering.

**window** [IWindow](#)

The window representing the rendering context.

**capacity** [uint](#)

Optional. The initial capacity of the vertex buffer.

# Properties

## Capacity

Specifies the maximum number of sprites that the PrimitiveBatch can process in a single draw call.

```
public uint Capacity { get; }
```

### Property Value

[uint](#)

## DrawCallCount

Gets the number of draw calls made during the current batch rendering session. This count is reset to zero each time [Begin\(CommandList, OutputDescription, Effect?, BlendStateDescription?, DepthStencilStateDescription?, RasterizerStateDescription?, Matrix4x4?, Matrix4x4?\)](#) is called and increments with each call to Bliss.CSharp.Graphics.Rendering.Batches.PrimitiveBatch.Flush().

```
public int DrawCallCount { get; }
```

### Property Value

[int](#)

## GraphicsDevice

Represents the graphics device used for rendering operations within the [PrimitiveBatch](#) class. The [GraphicsDevice](#) is used to create resources, manage rendering pipelines, and issue draw commands.

```
public GraphicsDevice GraphicsDevice { get; }
```

### Property Value

[GraphicsDevice](#)

# Window

Represents the window used for rendering graphics.

```
public IWindow Window { get; }
```

Property Value

[IWindow](#)

## Methods

### AddVertices(List<PrimitiveVertex2D>)

Adds a collection of vertices to the current batch for rendering.

```
public void AddVertices(List<PrimitiveVertex2D> vertices)
```

Parameters

`vertices` [List](#)<[PrimitiveVertex2D](#)>

The list of vertices to be added to the batch.

### Begin(CommandList, OutputDescription, Effect?, BlendStateDescription?, DepthStencilStateDescription?, RasterizerStateDescription?, Matrix4x4?, Matrix4x4?)

Begins a new batch of primitive drawing operations with specified rendering configurations.

```
public void Begin(CommandList commandList, OutputDescription output, Effect? effect = null,  
BlendStateDescription? blendState = null, DepthStencilStateDescription? depthStencilState =  
null, RasterizerStateDescription? rasterizerState = null, Matrix4x4? projection = null,  
Matrix4x4? view = null)
```

Parameters

## commandList [CommandList](#)

The command list to record drawing commands.

## output [OutputDescription](#)

The output description defining the render target configuration.

## effect [Effect](#)

Optional. The effect to use for rendering operations. Defaults to the global default primitive effect if not specified.

## blendState [BlendStateDescription](#)?

Optional. The blend state description used for rendering. Defaults to a single alpha blend if not specified.

## depthStencilState [DepthStencilStateDescription](#)?

Optional. The depth stencil state description used for rendering. Defaults to disabled depth-stencil testing if not specified.

## rasterizerState [RasterizerStateDescription](#)?

Optional. The rasterizer state description used for rendering. Defaults to cull none if not specified.

## projection [Matrix4x4](#)?

Optional. The projection matrix for the rendering. Defaults to an orthographic projection matrix if not specified.

## view [Matrix4x4](#)?

Optional. The view matrix for the rendering. Defaults to the identity matrix if not specified.

## Exceptions

### [Exception](#)

Thrown when the method is called before the previous batch has been properly ended.

## Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

## Parameters

**disposing** [bool](#)

True if called from user code; false if called from a finalizer.

## DrawEmptyCircle(Vector2, float, int, int, Color?)

Draws an empty circle at the specified position with the given radius, thickness, and number of segments.

```
public void DrawEmptyCircle(Vector2 position, float radius, int thickness, int segments,  
Color? color = null)
```

## Parameters

**position** [Vector2](#)

The center position of the circle.

**radius** [float](#)

The radius of the circle.

**thickness** [int](#)

The thickness of the circle's outline.

**segments** [int](#)

The number of segments to divide the circle into. Must be at least 4.

**color** [Color](#)?

The color of the circle's outline. Defaults to white if not specified.

## DrawEmptyCircleSector(Vector2, float, float, float, int, int, Color?)

Draws an empty circle sector using the specified parameters.

```
public void DrawEmptyCircleSector(Vector2 position, float radius, float startAngle, float endAngle, int thickness, int segments, Color? color = null)
```

## Parameters

**position** [Vector2](#)

The position of the center of the circle.

**radius** [float](#)

The radius of the circle sector.

**startAngle** [float](#)

The starting angle of the sector in degrees.

**endAngle** [float](#)

The ending angle of the sector in degrees.

**thickness** [int](#)

The thickness of the circle sector line.

**segments** [int](#)

The number of segments used to draw the sector. Minimum value is 4.

**color** [Color?](#)

The color of the circle sector line. Defaults to white if null.

## DrawEmptyEllipse(Vector2, Vector2, int, int, Color?)

Draws an empty ellipse at the specified position with the given radius, thickness, and number of segments.

```
public void DrawEmptyEllipse(Vector2 position, Vector2 radius, int thickness, int segments, Color? color = null)
```

## Parameters

**position** [Vector2](#)

The center position of the ellipse in 2D space.

**radius** [Vector2](#)

The radius of the ellipse along the X and Y axes.

**thickness** [int](#)

The thickness of the ellipse outline.

**segments** [int](#)

The number of segments to use for drawing the ellipse. Minimum value is 4.

**color** [Color?](#)

The color to use for the ellipse outline. If not specified, defaults to white.

## DrawEmptyRectangle(RectangleF, float, Vector2?, float, Color?)

Draws an empty rectangle with the specified dimensions, outline size, origin point, rotation, and color.

```
public void DrawEmptyRectangle(RectangleF rectangle, float thickness, Vector2? origin = null, float rotation = 0, Color? color = null)
```

## Parameters

**rectangle** [RectangleF](#)

Specifies the position and size of the rectangle.

**thickness** [float](#)

Width of the rectangle's outline.

**origin** [Vector2](#)?

Optional origin point for rotation and positioning. Defaults to (0,0).

**rotation** [float](#)

Optional rotation angle in degrees. Defaults to 0.0F.

#### color [Color?](#)

Optional color for the rectangle's outline. Defaults to white.

## DrawEmptyRing(Vector2, float, float, int, int, Color?)

Draws an empty ring at the specified position with given inner and outer radii, thickness, segments, and optional color.

```
public void DrawEmptyRing(Vector2 position, float innerRadius, float outerRadius, int thickness, int segments, Color? color = null)
```

### Parameters

#### position [Vector2](#)

The position where the ring will be drawn.

#### innerRadius [float](#)

The inner radius of the ring.

#### outerRadius [float](#)

The outer radius of the ring.

#### thickness [int](#)

The thickness of the ring.

#### segments [int](#)

The number of segments to use for drawing the ring. Minimum is 4.

#### color [Color?](#)

Optional color to use for drawing the ring. Defaults to white if not provided.

## DrawEmptyTriangle(Vector2, Vector2, Vector2, int, Color?)

Draws an empty triangle between the specified points with a given thickness and color.

```
public void DrawEmptyTriangle(Vector2 point1, Vector2 point2, Vector2 point3, int thickness,  
Color? color = null)
```

## Parameters

**point1** [Vector2](#)

The first vertex of the triangle.

**point2** [Vector2](#)

The second vertex of the triangle.

**point3** [Vector2](#)

The third vertex of the triangle.

**thickness** [int](#)

The thickness of the triangle edges.

**color** [Color?](#)

The color of the triangle edges. Defaults to white if not specified.

## DrawFilledCircle(Vector2, float, int, Color?)

Draws a filled circle at the specified position with the given radius, number of segments, and optional color.

```
public void DrawFilledCircle(Vector2 position, float radius, int segments, Color? color  
= null)
```

## Parameters

**position** [Vector2](#)

The position of the center of the circle.

**radius** [float](#)

The radius of the circle.

#### segments [int](#)

The number of segments to use for drawing the circle.

#### color [Color?](#)

The optional color of the circle. If null, defaults to white.

## DrawFilledCircleSector(Vector2, float, float, float, int, Color?)

Draws a filled sector of a circle on the screen at the specified position with the given parameters.

```
public void DrawFilledCircleSector(Vector2 position, float radius, float startAngle, float endAngle, int segments, Color? color = null)
```

### Parameters

#### position [Vector2](#)

The center position of the circle sector.

#### radius [float](#)

The radius of the circle sector.

#### startAngle [float](#)

The starting angle of the sector in degrees.

#### endAngle [float](#)

The ending angle of the sector in degrees.

#### segments [int](#)

Number of segments to use for drawing the sector.

#### color [Color?](#)

Optional color to use for the sector. Defaults to white if null.

## DrawFilledEllipse(Vector2, Vector2, int, Color?)

Draws a filled ellipse at the specified position with the given radius, number of segments, and optional color.

```
public void DrawFilledEllipse(Vector2 position, Vector2 radius, int segments, Color? color = null)
```

### Parameters

**position** [Vector2](#)

The center position of the ellipse.

**radius** [Vector2](#)

The horizontal and vertical radii of the ellipse.

**segments** [int](#)

The number of segments to divide the ellipse into.

**color** [Color](#)?

The color used to fill the ellipse. Defaults to white if not specified.

## DrawFilledRectangle(RectangleF, Vector2?, float, Color?)

Draws a rectangle with optional origin point, rotation, and color.

```
public void DrawFilledRectangle(RectangleF rectangle, Vector2? origin = null, float rotation = 0, Color? color = null)
```

### Parameters

**rectangle** [RectangleF](#)

The rectangle specifying the position and size.

**origin** [Vector2](#)?

Optional origin point for the rectangle, defaults to the top-left corner.

## `rotation` [float](#)

Optional rotation angle in radians, defaults to 0.0F.

## `color` [Color?](#)

Optional color for the rectangle, defaults to White.

# DrawFilledRing(`Vector2`, `float`, `float`, `int`, `Color?`)

Draws a filled ring at the specified position with the given inner and outer radii, segment count, and optional color.

```
public void DrawFilledRing(Vector2 position, float innerRadius, float outerRadius, int segments, Color? color = null)
```

## Parameters

### `position` [Vector2](#)

The center position of the ring.

### `innerRadius` [float](#)

The inner radius of the ring.

### `outerRadius` [float](#)

The outer radius of the ring.

### `segments` [int](#)

The number of segments to use for drawing the ring.

### `color` [Color?](#)

The color of the ring. If not provided, defaults to white.

# DrawFilledTriangle(`Vector2`, `Vector2`, `Vector2`, `Color?`)

Draws a filled triangle using the specified vertices and an optional color.

```
public void DrawFilledTriangle(Vector2 point1, Vector2 point2, Vector2 point3, Color? color = null)
```

## Parameters

**point1** [Vector2](#)

The first vertex of the triangle.

**point2** [Vector2](#)

The second vertex of the triangle.

**point3** [Vector2](#)

The third vertex of the triangle.

**color** [Color?](#)

The color of the triangle. If null, the default color is white.

## DrawLine(Vector2, Vector2, float, Color?)

Draws a line between two points with the specified thickness and color.

```
public void DrawLine(Vector2 start, Vector2 end, float thickness, Color? color = null)
```

## Parameters

**start** [Vector2](#)

The start point of the line.

**end** [Vector2](#)

The end point of the line.

**thickness** [float](#)

The thickness of the line. Default is 1.0.

**color** [Color?](#)

The color of the line. If null, defaults to white.

## End()

Ends the current batch of primitive drawing operations.

```
public void End()
```

Exceptions

[Exception ↗](#)

Thrown when the method is called before calling Begin().

## GetCurrentBlendState()

Retrieves the current blend state configuration used for rendering operations.

```
public BlendStateDescription GetCurrentBlendState()
```

Returns

[BlendStateDescription ↗](#)

The current blend state configuration as a [BlendStateDescription ↗](#).

Exceptions

[Exception ↗](#)

Thrown if the primitive batch operation has not been started.

## GetCurrentDepthStencilState()

Retrieves the current depth-stencil state configuration used for rendering.

```
public DepthStencilStateDescription GetCurrentDepthStencilState()
```

Returns

### [DepthStencilStateDescription](#)

The current [DepthStencilStateDescription](#) being used in the rendering pipeline.

Exceptions

### [Exception](#)

Thrown if the primitive batch operation has not been started.

## GetCurrentEffect()

Retrieves the currently active effect for the PrimitiveBatch.

```
public Effect GetCurrentEffect()
```

Returns

### [Effect](#)

The [Effect](#) that is currently being used by the PrimitiveBatch.

Exceptions

### [Exception](#)

Thrown if the primitive batch operation has not been started.

## GetCurrentOutput()

Retrieves the current output description for the primitive batch.

```
public OutputDescription GetCurrentOutput()
```

Returns

### [OutputDescription](#)

The current [OutputDescription](#) associated with the batch.

## Exceptions

### [Exception](#)

Thrown if the primitive batch operation has not been started.

## GetCurrentProjection()

Retrieves the current projection matrix being used for rendering operations.

```
public Matrix4x4 GetCurrentProjection()
```

## Returns

### [Matrix4x4](#)

The current projection as a [Matrix4x4](#).

## Exceptions

### [Exception](#)

Thrown if the primitive batch operation has not been started.

## GetCurrentRasterizerState()

Retrieves the current rasterizer state description used for rendering operations.

```
public RasterizerStateDescription GetCurrentRasterizerState()
```

## Returns

### [RasterizerStateDescription](#)

The current [RasterizerStateDescription](#) instance.

## Exceptions

## [Exception](#)

Thrown if the primitive batch operation has not been started.

## GetCurrentView()

Retrieves the current view matrix being used for rendering operations.

```
public Matrix4x4 GetCurrentView()
```

Returns

## [Matrix4x4](#)

The current [Matrix4x4](#) view matrix.

Exceptions

## [Exception](#)

Thrown if the primitive batch operation has not been started.

## ResetSettings()

Resets the [PrimitiveBatch](#) to default settings.

```
public void ResetSettings()
```

Exceptions

## [Exception](#)

Thrown if the primitive batch operation has not been started.

## SetBlendState(BlendStateDescription?)

Sets the blend state description to be used for subsequent rendering operations.

```
public void SetBlendState(BlendStateDescription? blendState)
```

## Parameters

**blendState** [BlendStateDescription](#)?

The blend state description to apply, or null to reset to the default single alpha blend state.

## Exceptions

[Exception](#)

Thrown if the primitive batch operation has not been started.

## SetDepthStencilState(DepthStencilStateDescription?)

Sets the depth-stencil state for rendering operations.

```
public void SetDepthStencilState(DepthStencilStateDescription? depthStencilState)
```

## Parameters

**depthStencilState** [DepthStencilStateDescription](#)?

The depth-stencil state to be applied. If null, the default disabled state is used.

## Exceptions

[Exception](#)

Thrown if the primitive batch operation has not been started.

## SetEffect(Effect?)

Sets the effect to be used when rendering primitives.

```
public void SetEffect(Effect? effect)
```

## Parameters

### **effect** [Effect](#)

The effect to be used. If null, the default primitive effect is used.

## Exceptions

### [Exception](#)

Thrown if the primitive batch operation has not been started.

## SetOutput(OutputDescription?)

Sets the rendering output for the PrimitiveBatch.

```
public void SetOutput(OutputDescription? output)
```

## Parameters

### **output** [OutputDescription](#)?

The output description to be used for rendering. If null, the main output will be used.

## Exceptions

### [Exception](#)

Thrown if the primitive batch operation has not been started.

## SetProjection(Matrix4x4?)

Sets the projection matrix to be used for rendering.

```
public void SetProjection(Matrix4x4? projection)
```

## Parameters

### **projection** [Matrix4x4](#)?

The new projection matrix. If null, a default orthographic projection is applied.

## Exceptions

### [Exception ↗](#)

Thrown if the primitive batch operation has not been started.

## SetRasterizerState(RasterizerStateDescription?)

Sets the current rasterizer state for rendering operations.

```
public void SetRasterizerState(RasterizerStateDescription? rasterizerState)
```

## Parameters

### [rasterizerState RasterizerStateDescription ↗?](#)

The rasterizer state description to apply. If null, a default state with no culling will be used.

## Exceptions

### [Exception ↗](#)

Thrown if the primitive batch operation has not been started.

## SetView(Matrix4x4?)

Sets the view matrix for the current rendering context.

```
public void SetView(Matrix4x4? view)
```

## Parameters

### [view Matrix4x4 ↗?](#)

The view matrix to be applied. If null, the identity matrix is used.

## Exceptions

## Exception

Thrown if the primitive batch operation has not been started.

# Namespace Bliss.CSharp.Graphics.Rendering. Batches.Sprites

## Classes

[SpriteBatch](#)

## Enums

[SpriteFlip](#)

# Class SpriteBatch

Namespace: [Bliss.CSharp.Graphics.Rendering.Batches.Sprites](#)

Assembly: Bliss.dll

```
public class SpriteBatch : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← SpriteBatch

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Constructors

## SpriteBatch(GraphicsDevice, IWindow, uint)

Initializes a new instance of the [SpriteBatch](#) class for batching and rendering 2D sprites.

```
public SpriteBatch(GraphicsDevice graphicsDevice, IWindow window, uint capacity = 15360)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The [GraphicsDevice](#) used for rendering.

**window** [IWindow](#)

The [IWindow](#) associated with the rendering context.

**capacity** [uint](#)

The maximum number of sprites the batch can hold. Defaults to 15,360.

# Properties

## Capacity

Specifies the maximum number of sprites that the SpriteBatch can process in a single draw call.

```
public uint Capacity { get; }
```

### Property Value

[uint](#)

## DrawCallCount

Gets the number of draw calls made during the current batch rendering session. This count is reset to zero each time [Begin\(CommandList, OutputDescription, Sampler?, Effect?, BlendStateDescription?, DepthStencilStateDescription?, RasterizerStateDescription?, Matrix4x4?, Matrix4x4?\)](#) is called and increments with each call to Bliss.CSharp.Graphics.Rendering.Batches.SpriteBatch.Flush().

```
public int DrawCallCount { get; }
```

### Property Value

[int](#)

## GraphicsDevice

Gets the [GraphicsDevice](#) associated with the [SpriteBatch](#). This device is responsible for managing and rendering graphics resources such as buffers, shaders, and textures.

```
public GraphicsDevice GraphicsDevice { get; }
```

### Property Value

[GraphicsDevice](#)

# Window

Represents the window used for rendering graphics.

```
public IWindow Window { get; }
```

Property Value

[IWindow](#)

## Methods

**AddQuad(Texture2D, SpriteVertex2D, SpriteVertex2D, SpriteVertex2D, SpriteVertex2D)**

Adds a quad to the sprite batch for rendering.

```
public void AddQuad(Texture2D texture, SpriteVertex2D topLeft, SpriteVertex2D topRight, SpriteVertex2D bottomLeft, SpriteVertex2D bottomRight)
```

Parameters

**texture** [Texture2D](#)

The [Texture2D](#) to be used for the quad's rendering.

**topLeft** [SpriteVertex2D](#)

The [SpriteVertex2D](#) defining the top-left vertex of the quad.

**topRight** [SpriteVertex2D](#)

The [SpriteVertex2D](#) defining the top-right vertex of the quad.

**bottomLeft** [SpriteVertex2D](#)

The [SpriteVertex2D](#) defining the bottom-left vertex of the quad.

**bottomRight** [SpriteVertex2D](#)

The [SpriteVertex2D](#) defining the bottom-right vertex of the quad.

## Exceptions

### [Exception ↗](#)

Thrown if the SpriteBatch has not been started by calling [Begin](#).

## [Begin\(CommandList, OutputDescription, Sampler?, Effect?, BlendStateDescription?, DepthStencilStateDescription?, RasterizerStateDescription?, Matrix4x4?, Matrix4x4?\)](#)

Begins a new sprite batch rendering session with the specified parameters.

```
public void Begin(CommandList commandList, OutputDescription output, Sampler? sampler = null, Effect? effect = null, BlendStateDescription? blendState = null, DepthStencilStateDescription? depthStencilState = null, RasterizerStateDescription? rasterizerState = null, Matrix4x4? projection = null, Matrix4x4? view = null)
```

## Parameters

### [commandList \[CommandList ↗\]\(#\)](#)

The [CommandList ↗](#) used to issue rendering commands.

### [output \[OutputDescription ↗\]\(#\)](#)

The [OutputDescription ↗](#) specifying the target render output configuration.

### [sampler \[Sampler ↗\]\(#\)](#)

The [Sampler ↗](#) defining sampler state for texture sampling. Defaults to a point sampler if not provided.

### [effect \[Effect\]\(#\)](#)

The [Effect](#) used for rendering sprites. Defaults to the global default sprite effect if not specified.

### [blendState \[BlendStateDescription ↗?\]\(#\)](#)

The [BlendStateDescription ↗](#) describing the blending mode. Defaults to single alpha blend if not specified.

### [depthStencilState \[DepthStencilStateDescription ↗?\]\(#\)](#)

The [DepthStencilStateDescription](#) specifying depth and stencil testing configuration. Defaults to disabled if not provided.

#### rasterizerState [RasterizerStateDescription](#)?

The [RasterizerStateDescription](#) defining rasterization settings. Defaults to no culling if not specified.

#### projection [Matrix4x4](#)?

The [Matrix4x4](#) representing the projection matrix. Defaults to an orthographic projection based on the window dimensions if not specified.

#### view [Matrix4x4](#)?

The [Matrix4x4](#) representing the view matrix. Defaults to the identity matrix if not specified.

## Exceptions

### [Exception](#)

Thrown when the method is called before the previous batch has been ended with a call to [End\(\)](#).

## Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

## Parameters

### [disposing](#) [bool](#)?

True if called from user code; false if called from a finalizer.

## DrawText(Font, string, Vector2, int, float, float, Vector2?, Vector2?, float, Color?, TextStyle, FontSystemEffect, int)

Draws the specified text at the given position with the provided font and styling options.

```
public void DrawText(Font font, string text, Vector2 position, int size, float characterSpacing = 0, float lineSpacing = 0, Vector2? scale = null, Vector2? origin = null,
```

```
float rotation = 0, Color? color = null, TextStyle style = TextStyle.None, FontSystemEffect effect = FontSystemEffect.None, int effectAmount = 0)
```

## Parameters

### font [Font](#)

The font to be used for drawing the text.

### text [string](#)

The text to be drawn.

### position [Vector2](#)

The position on the screen where the text will be drawn.

### size [int](#)

The size of the text.

### characterSpacing [float](#)

Optional spacing between characters. Default is 0.0F.

### lineSpacing [float](#)

Optional spacing between lines of text. Default is 0.0F.

### scale [Vector2](#)?

Optional scale applied to the text. Default is null.

### origin [Vector2](#)?

Optional origin point for rotation and scaling. Default is null.

### rotation [float](#)

Optional rotation angle in radians. Default is 0.0F.

### color [Color](#)?

Optional color of the text. Default is null.

### style [TextStyle](#)

Optional text style. Default is TextStyle.None.

#### effect FontSystemEffect

Optional effect applied to the text. Default is FontSystemEffect.None.

#### effectAmount [int](#)

Optional amount for the effect applied. Default is 0.

## DrawTexture(Texture2D, Vector2, Rectangle?, Vector2?, Vector2?, float, Color?, SpriteFlip)

Draws a texture using the specified parameters, including position, source rectangle, scale, origin, rotation, color, and flipping.

```
public void DrawTexture(Texture2D texture, Vector2 position, Rectangle? sourceRect = null,
Vector2? scale = null, Vector2? origin = null, float rotation = 0, Color? color = null,
SpriteFlip flip = SpriteFlip.None)
```

### Parameters

#### texture [Texture2D](#)

The texture to be drawn.

#### position [Vector2](#)

The position where the texture will be drawn.

#### sourceRect [Rectangle](#)?

The source rectangle within the texture. If null, the entire texture is used.

#### scale [Vector2](#)?

The scale factor for resizing the texture. If null, the texture is drawn at its original size.

#### origin [Vector2](#)?

The origin point for rotation and scaling. If null, the origin is set to the top-left corner.

#### rotation [float](#)

The rotation angle in radians.

#### color [Color](#)?

The color to tint the texture. If null, the texture is drawn with its original colors.

#### flip [SpriteFlip](#)

Specifies how the texture should be flipped horizontally or vertically.

## End()

Ends the current drawing session that was initiated by a call to [Begin\(CommandList, OutputDescription, Sampler?, Effect?, BlendStateDescription?, DepthStencilStateDescription?, RasterizerStateDescription?, Matrix4x4?, Matrix4x4?\)](#). This method finalizes the batch operations by flushing all pending draw calls.

```
public void End()
```

## Exceptions

### [Exception](#) ↗

Thrown if the SpriteBatch is in a state where [Begin\(CommandList, OutputDescription, Sampler?, Effect?, BlendStateDescription?, DepthStencilStateDescription?, RasterizerStateDescription?, Matrix4x4?, Matrix4x4?\)](#) was not called beforehand.

## GetCurrentBlendState()

Retrieves the current blend state used by the [SpriteBatch](#) for rendering operations.

```
public BlendStateDescription GetCurrentBlendState()
```

## Returns

### [BlendStateDescription](#) ↗

The [BlendStateDescription](#) representing the current blending configuration.

## Exceptions

## [Exception](#)

Thrown if the sprite batch operation has not been started.

## GetCurrentDepthStencilState()

Gets the current depth and stencil state configuration used for rendering in the [SpriteBatch](#).

```
public DepthStencilStateDescription GetCurrentDepthStencilState()
```

Returns

### [DepthStencilStateDescription](#)

The current [DepthStencilStateDescription](#) used by the [SpriteBatch](#) for rendering.

Exceptions

## [Exception](#)

Thrown if the sprite batch operation has not been started.

## GetCurrentEffect()

Retrieves the current [Effect](#) being used by the [SpriteBatch](#).

```
public Effect GetCurrentEffect()
```

Returns

### [Effect](#)

The active [Effect](#) instance used for rendering, or null if no effect is set.

Exceptions

## [Exception](#)

Thrown if the sprite batch operation has not been started.

## GetCurrentOutput()

Retrieves the current [OutputDescription](#) being used by the sprite batch.

```
public OutputDescription GetCurrentOutput()
```

Returns

[OutputDescription](#)

The [OutputDescription](#) currently associated with the sprite batch.

Exceptions

[Exception](#)

Thrown if the sprite batch operation has not been started.

## GetCurrentProjection()

Retrieves the current projection matrix used by the [SpriteBatch](#).

```
public Matrix4x4 GetCurrentProjection()
```

Returns

[Matrix4x4](#)

The current [Matrix4x4](#) projection matrix.

Exceptions

[Exception](#)

Thrown if the sprite batch operation has not been started.

## GetCurrentRasterizerState()

Gets the current rasterizer state used for configuring rasterization settings in the rendering pipeline.

```
public RasterizerStateDescription GetCurrentRasterizerState()
```

Returns

[RasterizerStateDescription](#)

A [RasterizerStateDescription](#) representing the current rasterizer state.

Exceptions

[Exception](#)

Thrown if the sprite batch operation has not been started.

## GetCurrentSampler()

Retrieves the current [Sampler](#) used for texture sampling operations in the [SpriteBatch](#).

```
public Sampler GetCurrentSampler()
```

Returns

[Sampler](#)

The current [Sampler](#) instance being used.

Exceptions

[Exception](#)

Thrown if the sprite batch operation has not been started.

## GetCurrentView()

Retrieves the current view matrix used for rendering.

```
public Matrix4x4 GetCurrentView()
```

Returns

## [Matrix4x4](#)

The current [Matrix4x4](#) view matrix.

Exceptions

## [Exception](#)

Thrown if the sprite batch operation has not been started.

## ResetSettings()

Resets the [SpriteBatch](#) to default settings.

```
public void ResetSettings()
```

Exceptions

## [Exception](#)

Thrown if the sprite batch operation has not been started.

## SetBlendState(BlendStateDescription?)

Updates the current blend state of the [SpriteBatch](#) for rendering sprites.

```
public void SetBlendState(BlendStateDescription? blendState)
```

Parameters

**blendState** [BlendStateDescription](#)?

The [BlendStateDescription](#) to set. If null, defaults to [SINGLE\\_ALPHA\\_BLEND](#).

Exceptions

## [Exception](#)

Thrown if the sprite batch operation has not been started.

## SetDepthStencilState(DepthStencilStateDescription?)

Sets the depth-stencil state to be used by the [SpriteBatch](#) during rendering operations.

```
public void SetDepthStencilState(DepthStencilStateDescription? depthStencilState)
```

### Parameters

**depthStencilState** [DepthStencilStateDescription](#)?

The [DepthStencilStateDescription](#) to use. If null, defaults to a disabled depth-stencil state.

### Exceptions

[Exception](#)

Thrown if the sprite batch operation has not been started.

## SetEffect(Effect?)

Sets the rendering effect to be used by the [SpriteBatch](#) during draw operations.

```
public void SetEffect(Effect? effect)
```

### Parameters

**effect** [Effect](#)

The [Effect](#) to set for rendering. If null, the default effect is used.

### Exceptions

[Exception](#)

Thrown if the sprite batch operation has not been started.

## SetOutput(OutputDescription?)

Sets the output description for the [SpriteBatch](#). If the specified output is null, the default main output is used instead.

```
public void SetOutput(OutputDescription? output)
```

### Parameters

**output** [OutputDescription](#)?

The optional [OutputDescription](#) to set. Defaults to the main output if null.

### Exceptions

[Exception](#)

Thrown if the sprite batch operation has not been started.

## SetProjection(Matrix4x4?)

Sets the projection matrix for rendering sprites.

```
public void SetProjection(Matrix4x4? projection)
```

### Parameters

**projection** [Matrix4x4](#)?

The [Matrix4x4](#) to use as the projection matrix. If null, an orthographic projection will be created based on the window dimensions.

### Exceptions

[Exception](#)

Thrown if the sprite batch operation has not been started.

## SetRasterizerState(RasterizerStateDescription?)

Updates the current rasterizer state of the [SpriteBatch](#) to the specified value.

```
public void SetRasterizerState(RasterizerStateDescription? rasterizerState)
```

## Parameters

rasterizerState [RasterizerStateDescription](#)?

The new [RasterizerStateDescription](#) to be applied. If null, the default rasterizer state [CULL\\_NONE](#) will be used.

## Exceptions

[Exception](#)

Thrown if the sprite batch operation has not been started.

## SetSampler(Sampler?)

Updates the current sampler state for the [SpriteBatch](#) to the specified value.

```
public void SetSampler(Sampler? sampler)
```

## Parameters

sampler [Sampler](#)

The new [Sampler](#) to use, or null to reset to the default sampler.

## Exceptions

[Exception](#)

Thrown if the sprite batch operation has not been started.

## SetView(Matrix4x4?)

Sets the current view matrix for rendering.

```
public void SetView(Matrix4x4? view)
```

## Parameters

**view** [Matrix4x4](#)?

The [Matrix4x4](#) view matrix to apply. If null, the identity matrix will be used.

## Exceptions

[Exception](#)

Thrown if the sprite batch operation has not been started.

# Enum SpriteFlip

Namespace: [Bliss.CSharp.Graphics.Rendering.Batches.Sprites](#)

Assembly: Bliss.dll

```
public enum SpriteFlip
```

## Fields

**Both** = 3

The sprite is flipped both vertically and horizontally.

**Horizontal** = 2

The sprite is flipped horizontally.

**None** = 0

No flipping applied to the sprite.

**Vertical** = 1

The sprite is flipped vertically.

# Namespace Bliss.CSharp.Graphics.Rendering. Passes

## Classes

[FullScreenRenderPass](#)

# Class FullScreenRenderPass

Namespace: [Bliss.CSharp.Graphics.Rendering.Passes](#)

Assembly: Bliss.dll

```
public class FullScreenRenderPass : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← FullScreenRenderPass

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### FullScreenRenderPass(GraphicsDevice)

Initializes a new instance of the [FullScreenRenderPass](#) class, setting up the necessary buffers and pipeline for full-screen rendering.

```
public FullScreenRenderPass(GraphicsDevice graphicsDevice)
```

## Parameters

graphicsDevice [GraphicsDevice](#)

The graphics device used for resource creation and rendering.

## Properties

### GraphicsDevice

The graphics device used for rendering.

```
public GraphicsDevice GraphicsDevice { get; }
```

Property Value

[GraphicsDevice](#)

## Methods

### Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

Parameters

[disposing](#) [bool](#)

True if called from user code; false if called from a finalizer.

### Draw(CommandList, RenderTexture2D, OutputDescription, Effect?, Sampler?, BlendStateDescription?, DepthStencilStateDescription?, RasterizerStateDescription?)

Executes the draw operation using the specified resources, rendering configurations, and GPU states.

```
public void Draw(CommandList commandList, RenderTexture2D renderTexture, OutputDescription  
output, Effect? effect = null, Sampler? sampler = null, BlendStateDescription? blendState =  
null, DepthStencilStateDescription? depthStencilState = null, RasterizerStateDescription?  
rasterizerState = null)
```

Parameters

[commandList](#) [CommandList](#)

The command list for issuing draw commands to the graphics device.

**renderTexture** [RenderTexture2D](#)

The render texture used as the input or output target for rendering operations.

**output** [OutputDescription](#)

The output description detailing the format and layout of render targets and depth-stencil buffers.

**effect** [Effect](#)

An optional shader effect utilized for rendering. A default effect is applied if none is specified.

**sampler** [Sampler](#)

An optional sampler used for texture sampling in the rendering process. If not set, a default point sampler is used.

**blendState** [BlendStateDescription](#)?

An optional blend state configuration for blending operations. Defaults to alpha blending if not provided.

**depthStencilState** [DepthStencilStateDescription](#)?

An optional depth-stencil state description to control depth and stencil testing. A disabled state is used by default.

**rasterizerState** [RasterizerStateDescription](#)?

An optional rasterizer state description to configure rasterization settings. Defaults to a standard rasterizer configuration if not specified.

# Namespace Bliss.CSharp.Graphics.Rendering. Renderers

## Classes

[ImmediateRenderer](#)

# Class ImmediateRenderer

Namespace: [Bliss.CSharp.Graphics.Rendering.Renderers](#)

Assembly: Bliss.dll

```
public class ImmediateRenderer : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← ImmediateRenderer

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Constructors

## ImmediateRenderer(GraphicsDevice, uint)

Initializes a new instance of the [ImmediateRenderer](#) class with the specified graphics device, output, effect, and capacity.

```
public ImmediateRenderer(GraphicsDevice graphicsDevice, uint capacity = 30720)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device used for rendering.

**capacity** [uint](#)

The maximum number of vertices that can be batched. Defaults to 30720.

# Properties

## Capacity

Gets the maximum number of vertices that can be batched.

```
public uint Capacity { get; }
```

### Property Value

[uint](#)

## GraphicsDevice

Gets the graphics device used for rendering.

```
public GraphicsDevice GraphicsDevice { get; }
```

### Property Value

[GraphicsDevice](#)

# Methods

## Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

### Parameters

**disposing** [bool](#)

True if called from user code; false if called from a finalizer.

## DrawBillboard(CommandList, OutputDescription, Vector3, Vector2?, Color?)

Draws a billboard at the specified position with optional scaling and color parameters.

```
public void DrawBillboard(CommandList commandList, OutputDescription output, Vector3  
position, Vector2? scale = null, Color? color = null)
```

### Parameters

`commandList` [CommandList](#)

The command list used for rendering commands.

`output` [OutputDescription](#)

The output description that defines rendering target properties.

`position` [Vector3](#)

The 3D position where the billboard will be rendered.

`scale` [Vector2](#)?

The optional scale factor for the billboard. Defaults to a scale of 1 if not specified.

`color` [Color](#)?

The optional color of the billboard. Defaults to white if not specified.

## DrawBoundingBox(CommandList, OutputDescription, Transform, BoundingBox, Color?)

Draws the edges of the specified bounding box using the given transformation and color.

```
public void DrawBoundingBox(CommandList commandList, OutputDescription output, Transform  
transform, BoundingBox box, Color? color = null)
```

### Parameters

`commandList` [CommandList](#)

The command list used for issuing rendering commands.

#### output [OutputDescription](#)

The output description for the rendering target.

#### transform [Transform](#)

The transformation to apply to the bounding box before rendering.

#### box [BoundingBox](#)

The bounding box to be drawn.

#### color [Color?](#)

The color to use for the bounding box edges. If null, white is used as the default color.

## DrawCapsule(CommandList, OutputDescription, Transform, float, float, int, Color?)

Renders a 3D capsule using the specified transformation, dimensions, and visual properties.

```
public void DrawCapsule(CommandList commandList, OutputDescription output, Transform  
transform, float radius, float height, int slices, Color? color = null)
```

## Parameters

#### commandList [CommandList](#)

The command list used to issue draw commands.

#### output [OutputDescription](#)

The output description for the render target.

#### transform [Transform](#)

The transformation to apply to the capsule, including position, rotation, and scale.

#### radius [float](#)

The radius of the capsule's hemispherical ends and cylindrical body.

### `height` [float](#)

The total height of the capsule.

### `slices` [int](#)

The number of subdivisions around the circumference of the capsule. Must be 3 or greater.

### `color` [Color?](#)

The color of the capsule. If null, defaults to white.

## DrawCapsuleWires(CommandList, OutputDescription, Transform, float, float, int, Color?)

Draws the wireframe of a capsule based on the specified transform, dimensions, and color.

```
public void DrawCapsuleWires(CommandList commandList, OutputDescription output, Transform  
transform, float radius, float height, int slices, Color? color = null)
```

### Parameters

#### `commandList` [CommandList](#)

The command list used for issuing draw commands.

#### `output` [OutputDescription](#)

The output description containing information about the render target and configurations.

#### `transform` [Transform](#)

The transformation applied to the capsule, such as position, rotation, and scale.

#### `radius` [float](#)

The radius of the capsule's hemispherical ends and cylindrical body.

#### `height` [float](#)

The height of the cylindrical body of the capsule.

#### `slices` [int](#)

The number of divisions for rendering the capsule's rounded surface. Must be at least 3.

#### color [Color?](#)

The optional color of the capsule wireframe. Defaults to white if not specified.

## DrawCone(CommandList, OutputDescription, Transform, float, float, int, Color?)

Draws a 3D cone using the specified parameters.

```
public void DrawCone(CommandList commandList, OutputDescription output, Transform transform,
float radius, float height, int slices, Color? color = null)
```

### Parameters

#### commandList [CommandList](#)

The command list to record drawing commands.

#### output [OutputDescription](#)

The output description specifying the target rendering surface.

#### transform [Transform](#)

The transform specifying the position, rotation, and scale of the cone in world space.

#### radius [float](#)

The radius of the base of the cone.

#### height [float](#)

The height of the cone from its base to its apex.

#### slices [int](#)

The number of slices used to construct the cone's base. Must be at least 3.

#### color [Color?](#)

An optional parameter to define the color of the cone. Defaults to white if not provided.

## DrawConeWires(CommandList, OutputDescription, Transform, float, float, int, Color?)

Draws a wireframe representation of a cone in 3D space.

```
public void DrawConeWires(CommandList commandList, OutputDescription output, Transform transform, float radius, float height, int slices, Color? color = null)
```

### Parameters

`commandList` [CommandList](#)

The command list used to issue rendering commands.

`output` [OutputDescription](#)

The output description that specifies render target details.

`transform` [Transform](#)

The transformation to apply to the cone, including position, rotation, and scale.

`radius` [float](#)

The radius of the cone's base.

`height` [float](#)

The height of the cone from the base to its tip.

`slices` [int](#)

The number of slices used to approximate the circular base. Must be at least 3.

`color` [Color?](#)

An optional color for the wireframe. Defaults to white if null.

## DrawCube(CommandList, OutputDescription, Transform, Vector3, Color?)

Draws a cube with the specified command list, output description, transformation, size, and optional color.

```
public void DrawCube(CommandList commandList, OutputDescription output, Transform transform,  
Vector3 size, Color? color = null)
```

## Parameters

**commandList** [CommandList](#)

The command list used for submitting rendering commands.

**output** [OutputDescription](#)

The output description specifying the render target details.

**transform** [Transform](#)

The transformation to be applied to the cube.

**size** [Vector3](#)

The size of the cube in 3D space.

**color** [Color](#)?

An optional color for the cube; if null, white is used.

## DrawCubeWires(CommandList, OutputDescription, Transform, Vector3, Color?)

Draws the wireframe of a cube with the specified transform, size, and optional color.

```
public void DrawCubeWires(CommandList commandList, OutputDescription output, Transform  
transform, Vector3 size, Color? color = null)
```

## Parameters

**commandList** [CommandList](#)

The command list used to issue rendering commands.

**output** [OutputDescription](#)

The output description that specifies the target render output.

## transform [Transform](#)

The transformation to apply to the cube, including position, rotation, and scale.

## size [Vector3](#)

The dimensions of the cube to be drawn.

## color [Color?](#)

An optional color for the wireframe. If not provided, the default color is white.

# DrawCylinder(CommandList, OutputDescription, Transform, float, float, int, Color?)

Renders a cylinder using the specified command list, output description, transform, dimensions, slice count, and optional color.

```
public void DrawCylinder(CommandList commandList, OutputDescription output, Transform  
transform, float radius, float height, int slices, Color? color = null)
```

## Parameters

### commandList [CommandList](#)

The command list used for issuing rendering commands.

### output [OutputDescription](#)

The output description that defines the rendering target.

### transform [Transform](#)

The transform used to position and orient the cylinder in 3D space.

### radius [float](#)

The radius of the cylinder's top and bottom caps.

### height [float](#)

The height of the cylinder from bottom to top cap.

### `slices` [int](#)

The number of segments used to approximate the cylindrical surface. Minimum value is 3.

### `color` [Color?](#)

The optional color of the cylinder. If null, a default white color is applied.

## DrawCylinderWires(CommandList, OutputDescription, Transform, float, float, int, Color?)

Draws the wireframe representation of a cylinder using the specified transformation, radius, height, and number of slices.

```
public void DrawCylinderWires(CommandList commandList, OutputDescription output, Transform transform, float radius, float height, int slices, Color? color = null)
```

### Parameters

#### `commandList` [CommandList](#)

The command list used to issue rendering commands.

#### `output` [OutputDescription](#)

The output description that defines the rendering target.

#### `transform` [Transform](#)

The transformation applied to position, rotate, and scale the cylinder in the scene.

#### `radius` [float](#)

The radius of the cylinder.

#### `height` [float](#)

The height of the cylinder.

#### `slices` [int](#)

The number of divisions around the cylinder's circumference. Must be 3 or greater.

## color [Color?](#)

The color of the cylinder's wireframe. Defaults to white if not specified.

# DrawGird(CommandList, OutputDescription, Transform, int, int, Color?)

Renders a grid using the specified command list, output description, transformation matrix, slice count, spacing, and optional color.

```
public void DrawGird(CommandList commandList, OutputDescription output, Transform transform,  
int slices, int spacing, Color? color = null)
```

## Parameters

### commandList [CommandList](#)

The command list used for rendering the grid.

### output [OutputDescription](#)

The output description that defines the render target's properties.

### transform [Transform](#)

The transformation applied to the grid.

### slices [int](#)

The number of divisions (slices) in the grid. Must be greater than or equal to 1.

### spacing [int](#)

The distance (spacing) between each grid line. Must be greater than or equal to 1.

### color [Color?](#)

An optional color for the grid lines. Defaults to white if not specified.

# DrawHemisphere(CommandList, OutputDescription, Transform, float, int, int, Color?)

Renders a 3D hemisphere with the specified transformation, radius, rings, slices, and optional color.

```
public void DrawHemisphere(CommandList commandList, OutputDescription output, Transform transform, float radius, int rings, int slices, Color? color = null)
```

## Parameters

**commandList** [CommandList](#)

The command list used for issuing draw commands.

**output** [OutputDescription](#)

The output description for the current render target.

**transform** [Transform](#)

The transformation to apply to the hemisphere.

**radius** [float](#)

The radius of the hemisphere.

**rings** [int](#)

The number of rings used to construct the hemisphere. Must be 3 or greater.

**slices** [int](#)

The number of slices used to construct the hemisphere. Must be 3 or greater.

**color** [Color?](#)

An optional color to apply to the hemisphere. If null, the default color will be white.

## DrawHemisphereWires(CommandList, OutputDescription, Transform, float, int, int, Color?)

Draws the wireframe outline of a hemisphere using the specified parameters.

```
public void DrawHemisphereWires(CommandList commandList, OutputDescription output, Transform transform, float radius, int rings, int slices, Color? color = null)
```

## Parameters

**commandList** [CommandList](#)

The command list used for rendering commands.

**output** [OutputDescription](#)

The output description that determines the rendering target.

**transform** [Transform](#)

The transformation to apply to the hemisphere.

**radius** [float](#)

The radius of the hemisphere.

**rings** [int](#)

The number of horizontal subdivisions (rings) for the hemisphere.

**slices** [int](#)

The number of vertical subdivisions (slices) for the hemisphere.

**color** [Color?](#)

An optional color for the hemisphere. If null, it defaults to white.

## DrawKnot(CommandList, OutputDescription, Transform, float, float, int, int, Color?)

Draws a knot shape with the specified parameters using a command list and defined properties such as transformations, radii, number of segments, and optional color.

```
public void DrawKnot(CommandList commandList, OutputDescription output, Transform transform,  
float radius, float tubeRadius, int radSeg, int sides, Color? color = null)
```

## Parameters

**commandList** [CommandList](#)

The command list used for rendering commands.

#### output [OutputDescription](#)

The output description specifying the rendering context.

#### transform [Transform](#)

The transformation to apply to the knot during rendering.

#### radius [float](#)

The overall radius of the knot.

#### tubeRadius [float](#)

The radius of the tube forming the knot structure.

#### radSeg [int](#)

The number of radial segments forming the knot. Must be 3 or greater.

#### sides [int](#)

The number of sides of the tube forming the knot. Must be 3 or greater.

#### color [Color?](#)

An optional color to apply to the knot. Defaults to white when null.

## DrawKnotWires(CommandList, OutputDescription, Transform, float, float, int, int, Color?)

Renders the wireframe of a knot shape using the specified transformation, radii, and segments.

```
public void DrawKnotWires(CommandList commandList, OutputDescription output, Transform transform, float radius, float tubeRadius, int radSeg, int sides, Color? color = null)
```

## Parameters

#### commandList [CommandList](#)

The command list used for issuing rendering commands.

**output** [OutputDescription](#)

The output description specifying the rendering target.

**transform** [Transform](#)

The transformation applied to the knot wireframe.

**radius** [float](#)

The radius of the knot.

**tubeRadius** [float](#)

The radius of the tube comprising the knot's wireframe.

**radSeg** [int](#)

The number of radial segments making up the knot. Minimum value is 3.

**sides** [int](#)

The number of sides for the tube cross-section. Minimum value is 3.

**color** [Color?](#)

An optional color to use for the wireframe; if null, the default is white.

## DrawLine(CommandList, OutputDescription, Vector3, Vector3, Color?)

Draws a line between two points in 3D space with a specified optional color.

```
public void DrawLine(CommandList commandList, OutputDescription output, Vector3 startPos, Vector3 endPos, Color? color = null)
```

### Parameters

**commandList** [CommandList](#)

The command list used for recording drawing commands.

**output** [OutputDescription](#)

The output description of the render target.

#### startPos [Vector3](#)

The starting position of the line in 3D space.

#### endPos [Vector3](#)

The ending position of the line in 3D space.

#### color [Color?](#)

The optional color of the line. If not provided, defaults to white.

## DrawSphere(CommandList, OutputDescription, Transform, float, int, int, Color?)

Draws a sphere with the specified transformation, radius, number of rings, slices, and optional color.

```
public void DrawSphere(CommandList commandList, OutputDescription output, Transform  
transform, float radius, int rings, int slices, Color? color = null)
```

### Parameters

#### commandList [CommandList](#)

The command list used for issuing rendering commands.

#### output [OutputDescription](#)

The output description determining the render target.

#### transform [Transform](#)

The transformation to be applied to the sphere.

#### radius [float](#)

The radius of the sphere.

#### rings [int](#)

The number of horizontal subdivisions of the sphere.

## slices [int](#)

The number of vertical subdivisions of the sphere.

## color [Color?](#)

An optional color for the sphere; defaults to white if not provided.

# DrawSphereWires(CommandList, OutputDescription, Transform, float, int, int, Color?)

Draws the wireframe of a sphere with the specified transform, radius, number of rings, slices, and optional color.

```
public void DrawSphereWires(CommandList commandList, OutputDescription output, Transform  
transform, float radius, int rings, int slices, Color? color = null)
```

## Parameters

### commandList [CommandList](#)

The command list used for issuing draw commands to the GPU.

### output [OutputDescription](#)

The output description of the render target where the sphere will be drawn.

### transform [Transform](#)

The transformation to apply to the sphere, including position, rotation, and scale.

### radius [float](#)

The radius of the sphere.

### rings [int](#)

The number of horizontal subdivisions (rings) of the sphere.

### slices [int](#)

The number of vertical subdivisions (slices) of the sphere.

`color` [Color?](#)

The optional color for the sphere's wireframe. Defaults to white if not specified.

## DrawTorus(CommandList, OutputDescription, Transform, float, float, int, int, Color?)

Renders a torus shape using the specified transformation, dimensions, and color.

```
public void DrawTorus(CommandList commandList, OutputDescription output, Transform transform, float radius, float size, int radSeg, int sides, Color? color = null)
```

### Parameters

`commandList` [CommandList](#)

The command list used for issuing rendering commands.

`output` [OutputDescription](#)

The output description specifying render target details.

`transform` [Transform](#)

The transformation to apply to the torus, including position, rotation, and scale.

`radius` [float](#)

The radius of the inner circle of the torus.

`size` [float](#)

The thickness of the torus.

`radSeg` [int](#)

The number of segments along the radial direction of the torus. Must be 3 or greater.

`sides` [int](#)

The number of sides to approximate the circular cross-section of the torus. Must be 3 or greater.

`color` [Color?](#)

The color of the torus. If null, the default color will be white.

## DrawTorusWires(CommandList, OutputDescription, Transform, float, float, int, int, Color?)

Renders a wireframe torus using the specified transformation, dimensions, and color.

```
public void DrawTorusWires(CommandList commandList, OutputDescription output, Transform transform, float radius, float size, int radSeg, int sides, Color? color = null)
```

### Parameters

`commandList` [CommandList](#)

The command list used for rendering the torus wireframe.

`output` [OutputDescription](#)

The output description used for the rendering pipeline.

`transform` [Transform](#)

The transformation applied to the torus, including position, rotation, and scale.

`radius` [float](#)

The radius of the inner circle of the torus.

`size` [float](#)

The thickness of the torus.

`radSeg` [int](#)

The number of radial segments. Must be 3 or greater.

`sides` [int](#)

The number of subdivisions around the circular cross-section. Must be 3 or greater.

`color` [Color?](#)

The optional color used for rendering the torus wireframe. Defaults to white if not specified.

## DrawVertices(CommandList, OutputDescription, Transform, List<ImmediateVertex3D>, List<uint>, PrimitiveTopology)

Draws a set of vertices using the specified command list, output description, transformation, vertex data, indices, and primitive topology.

```
public void DrawVertices(CommandList commandList, OutputDescription output, Transform transform, List<ImmediateVertex3D> vertices, List<uint> indices, PrimitiveTopology topology)
```

### Parameters

**commandList** [CommandList](#)

The command list used for issuing drawing commands.

**output** [OutputDescription](#)

The output description that defines how the rendered content is processed and displayed.

**transform** [Transform](#)

The transformation matrix to apply to the vertex data.

**vertices** [List](#)<[ImmediateVertex3D](#)>

The collection of vertices to render.

**indices** [List](#)<[uint](#)>

The collection of indices defining the order in which vertices are connected.

**topology** [PrimitiveTopology](#)

The primitive topology that specifies how the vertex data should be interpreted (e.g., triangle list, line strip).

## GetBlendState()

Retrieves the current blend state used by the renderer.

```
public BlendStateDescription GetBlendState()
```

Returns

### [BlendStateDescription](#)

The current [BlendStateDescription](#) instance.

## GetCurrentDepthStencilState()

Retrieves the currently bound depth stencil state description.

```
public DepthStencilStateDescription GetCurrentDepthStencilState()
```

Returns

### [DepthStencilStateDescription](#)

The current [DepthStencilStateDescription](#) instance being used.

## GetCurrentEffect()

Retrieves the current effect being used by the renderer.

```
public Effect GetCurrentEffect()
```

Returns

### [Effect](#)

The currently set [Effect](#) instance.

## GetCurrentRasterizerState()

Retrieves the current rasterizer state description used by the renderer.

```
public RasterizerStateDescription GetCurrentRasterizerState()
```

Returns

## [RasterizerStateDescription](#)

The currently bound rasterizer state.

## GetCurrentSampler()

Retrieves the currently active sampler used by the renderer.

```
public Sampler GetCurrentSampler()
```

Returns

### [Sampler](#)

The active [Sampler](#) instance.

## GetCurrentSourceRec()

Retrieves the current source rectangle used for rendering operations.

```
public Rectangle GetCurrentSourceRec()
```

Returns

### [Rectangle](#)

The current [Rectangle](#) source used for rendering.

## GetCurrentTexture()

Retrieves the currently active texture being used by the renderer.

```
public Texture2D GetCurrentTexture()
```

Returns

### [Texture2D](#)

The active [Texture2D](#) object currently bound to the renderer.

## ResetSettings()

Resets the [ImmediateRenderer](#) to default settings.

```
public void ResetSettings()
```

## SetBlendState(BlendStateDescription?)

Sets the current blend state for the renderer. If no blend state is provided, it defaults to [SINGLE ALPHA BLEND](#).

```
public void SetBlendState(BlendStateDescription? blendState)
```

### Parameters

`blendState` [BlendStateDescription](#)?

The blend state to apply. Defaults to [SINGLE ALPHA BLEND](#) if null.

## SetDepthStencilState(DepthStencilStateDescription?)

Sets the depth stencil state for the renderer. If no state is provided, a default state of depth-only less/equal is used.

```
public void SetDepthStencilState(DepthStencilStateDescription? depthStencilState)
```

### Parameters

`depthStencilState` [DepthStencilStateDescription](#)?

The depth stencil state to set. If null, defaults to `DepthStencilStateDescription.DEPTH_ONLY_LESS_EQUAL`.

## SetEffect(Effect?)

Sets the current rendering effect for the ImmediateRenderer. If the provided effect is null, it defaults to the global default immediate renderer effect.

```
public void SetEffect(Effect? effect)
```

Parameters

**effect** [Effect](#)

The effect to be used. If null, the default ImmediateRenderer effect will be used.

## SetRasterizerState(RasterizerStateDescription?)

Updates the current rasterizer state for the renderer. If no rasterizer state is provided, the default state is used.

```
public void SetRasterizerState(RasterizerStateDescription? rasterizerState)
```

Parameters

**rasterizerState** [RasterizerStateDescription](#)?

The rasterizer state to apply to the pipeline. If null, the default rasterizer state is used.

## SetTexture(Texture2D?, Sampler?, Rectangle?)

Sets the current texture, sampler, and optional source rectangle for rendering.

```
public void SetTexture(Texture2D? texture, Sampler? sampler = null, Rectangle? sourceRect  
= null)
```

Parameters

**texture** [Texture2D](#)

The texture to be used for rendering. If null, a default texture is used.

## sampler [Sampler](#)

The sampler to be applied to the texture. Defaults to a point sampler if null.

## sourceRect [Rectangle](#)?

The source rectangle specifying a portion of the texture to be used. If null, the entire texture is used.

# Namespace Bliss.CSharp.Graphics.VertexTypes

## Structs

[CubemapVertex3D](#)

[ImmediateVertex3D](#)

[PrimitiveVertex2D](#)

[SpriteVertex2D](#)

[Vertex3D](#)

# Struct CubemapVertex3D

Namespace: [Bliss.CSharp.Graphics.VertexTypes](#)

Assembly: Bliss.dll

```
public struct CubemapVertex3D
```

## Inherited Members

[ValueType.Equals\(object\)](#) , [ValueType.GetHashCode\(\)](#) , [ValueType.ToString\(\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### CubemapVertex3D(Vector3, Vector4)

Initializes a new instance of the [CubemapVertex3D](#) structure.

```
public CubemapVertex3D(Vector3 position, Vector4 color)
```

## Parameters

**position** [Vector3](#)

The position of the vertex in 3D space.

**color** [Vector4](#)

The color of the vertex.

## Fields

### Color

The color of the vertex.

```
public Vector4 Color
```

## Field Value

[Vector4](#) ↗

## Position

The position of the vertex in 3D space.

```
public Vector3 Position
```

## Field Value

[Vector3](#) ↗

## VertexLayout

Represents the layout description for the [CubemapVertex3D](#) structure.

```
public static VertexLayoutDescription VertexLayout
```

## Field Value

[VertexLayoutDescription](#) ↗

# Struct ImmediateVertex3D

Namespace: [Bliss.CSharp.Graphics.VertexTypes](#)

Assembly: Bliss.dll

```
public struct ImmediateVertex3D
```

## Inherited Members

[ValueType.Equals\(object\)](#) , [ValueType.GetHashCode\(\)](#) , [ValueType.ToString\(\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### ImmediateVertex3D(Vector3, Vector2, Vector4)

Initializes a new instance of the [ImmediateVertex3D](#) structure with the specified position, texture coordinates, and color.

```
public ImmediateVertex3D(Vector3 position, Vector2 texCoords, Vector4 color)
```

## Parameters

**position** [Vector3](#)

The position of the vertex in 3D space.

**texCoords** [Vector2](#)

The texture coordinates associated with the vertex.

**color** [Vector4](#)

The color of the vertex.

## Fields

### Color

The color of the vertex.

```
public Vector4 Color
```

Field Value

[Vector4](#)

## Position

The position of the vertex in 3D space.

```
public Vector3 Position
```

Field Value

[Vector3](#)

## TexCoords

The primary texture coordinates of the vertex.

```
public Vector2 TexCoords
```

Field Value

[Vector2](#)

## VertexLayout

Represents the layout description for the [ImmediateVertex3D](#) structure.

```
public static VertexLayoutDescription VertexLayout
```

Field Value

## VertexLayoutDescription

# Struct PrimitiveVertex2D

Namespace: [Bliss.CSharp.Graphics.VertexTypes](#)

Assembly: Bliss.dll

```
public struct PrimitiveVertex2D
```

## Inherited Members

[ValueType.Equals\(object\)](#) , [ValueType.GetHashCode\(\)](#) , [ValueType.ToString\(\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### PrimitiveVertex2D(Vector2, Vector4)

Initializes a new instance of the [PrimitiveVertex2D](#) struct with the specified position and color values.

```
public PrimitiveVertex2D(Vector2 position, Vector4 color)
```

## Parameters

**position** [Vector2](#)

The 2D position of the vertex.

**color** [Vector4](#)

The color of the vertex, represented as a Vector4 (RGBA).

## Fields

### Color

The color of the vertex.

```
public Vector4 Color
```

## Field Value

[Vector4](#) ↗

## Position

The position of the vertex in 2D space.

```
public Vector2 Position
```

## Field Value

[Vector2](#) ↗

## VertexLayout

Represents the layout description for the [PrimitiveVertex2D](#) structure.

```
public static VertexLayoutDescription VertexLayout
```

## Field Value

[VertexLayoutDescription](#) ↗

# Struct SpriteVertex2D

Namespace: [Bliss.CSharp.Graphics.VertexTypes](#)

Assembly: Bliss.dll

```
public struct SpriteVertex2D
```

## Inherited Members

[ValueType.Equals\(object\)](#) , [ValueType.GetHashCode\(\)](#) , [ValueType.ToString\(\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### SpriteVertex2D(Vector2, Vector2, Vector4)

Initializes a new instance of the [SpriteVertex2D](#) struct with the specified position, texture coordinates, and color.

```
public SpriteVertex2D(Vector2 position, Vector2 texCoords, Vector4 color)
```

## Parameters

**position** [Vector2](#)

The 2D position of the vertex.

**texCoords** [Vector2](#)

The texture coordinates of the vertex.

**color** [Vector4](#)

The color of the vertex as a vector with RGBA components.

## Fields

### Color

The color of the vertex.

```
public Vector4 Color
```

Field Value

[Vector4](#)

## Position

The position of the vertex in 2D space.

```
public Vector2 Position
```

Field Value

[Vector2](#)

## TexCoords

The texture coordinates of the vertex.

```
public Vector2 TexCoords
```

Field Value

[Vector2](#)

## VertexLayout

Represents the layout description for the [SpriteVertex2D](#) structure.

```
public static VertexLayoutDescription VertexLayout
```

Field Value

## [VertexLayoutDescription](#)

# Struct Vertex3D

Namespace: [Bliss.CSharp.Graphics.VertexTypes](#)

Assembly: Bliss.dll

```
public struct Vertex3D
```

## Inherited Members

[ValueType.Equals\(object\)](#) , [ValueType.GetHashCode\(\)](#) , [ValueType.ToString\(\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

**Vertex3D(Vector3, Vector4, UInt4, Vector2, Vector2, Vector3, Vector3, Vector4)**

Initializes a new instance of the [Vertex3D](#) struct with the specified position, bone weights, bone indices, texture coordinates, normal, tangent, and color values.

```
public Vertex3D(Vector3 position, Vector4 boneWeights, UInt4 boneIndices, Vector2 texCoords,  
Vector2 texCoords2, Vector3 normal, Vector3 tangent, Vector4 color)
```

## Parameters

**position** [Vector3](#)

The position of the vertex in 3D space.

**boneWeights** [Vector4](#)

The weights associated with bones for skeletal animation.

**boneIndices** [UInt4](#)

The indices of bones influencing this vertex.

**texCoords** [Vector2](#)

The primary texture coordinates of the vertex.

**texCoords2** [Vector2](#)

The secondary texture coordinates of the vertex.

**normal** [Vector3](#)

The normal vector at the vertex, used for lighting calculations.

**tangent** [Vector3](#)

The tangent vector at the vertex, used for normal mapping.

**color** [Vector4](#)

The color of the vertex, stored as an RGBA float vector.

## Fields

### BoneIndices

Represents the indices of the vertex's associated bones, used in conjunction with bone weights for skeletal animation.

**public** UInt4 BoneIndices

### Field Value

[UInt4](#)

### BoneWeights

Represents the weights of the vertex's associated bones, used for skinning in skeletal animation.

**public** Vector4 BoneWeights

### Field Value

[Vector4](#)

## Color

The color of the vertex.

```
public Vector4 Color
```

### Field Value

[Vector4](#)

## Normal

The normal vector of the vertex, used for lighting calculations.

```
public Vector3 Normal
```

### Field Value

[Vector3](#)

## Position

The position of the vertex in 3D space.

```
public Vector3 Position
```

### Field Value

[Vector3](#)

## Tangent

The tangent vector of the vertex, used for normal mapping.

```
public Vector3 Tangent
```

Field Value

[Vector3](#)

## TexCoords

The primary texture coordinates of the vertex.

```
public Vector2 TexCoords
```

Field Value

[Vector2](#)

## TexCoords2

The secondary texture coordinates of the vertex.

```
public Vector2 TexCoords2
```

Field Value

[Vector2](#)

## VertexLayout

Represents the layout description for the [Vertex3D](#) structure.

```
public static VertexLayoutDescription VertexLayout
```

Field Value

[VertexLayoutDescription](#)

## Methods

## AddBone(uint, float)

Adds a bone to the vertex and assigns a weight to it.

```
public void AddBone(uint id, float weight)
```

### Parameters

**id** [uint](#)

The identifier of the bone.

**weight** [float](#)

The weight of the bone influence.

# Namespace Bliss.CSharp.Images

## Classes

[AnimatedImage](#)

[Image](#)

# Class AnimatedImage

Namespace: [Bliss.CSharp.Images](#)

Assembly: Bliss.dll

```
public class AnimatedImage
```

## Inheritance

[object](#) ← AnimatedImage

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### AnimatedImage(Stream)

Initializes a new instance of the [AnimatedImage](#) class from a stream.

```
public AnimatedImage(Stream stream)
```

#### Parameters

**stream** [Stream](#)

The stream containing the animated image data.

### AnimatedImage(string)

Initializes a new instance of the [AnimatedImage](#) class from a file path.

```
public AnimatedImage(string path)
```

#### Parameters

## path [string](#)

The file path to the animated image (e.g., a GIF).

# Properties

## Frames

Gets a read-only dictionary containing the frames of the animated image and their respective delays (in milliseconds).

```
public IReadOnlyDictionary<Image, int> Frames { get; }
```

## Property Value

[IReadOnlyDictionary](#)<Image, int>

## SpriteSheet

Gets the sprite sheet representation of the animated image, where all frames are arranged in a single image.

```
public Image SpriteSheet { get; }
```

## Property Value

[Image](#)

# Methods

## GetFrameCount()

Retrieves the total number of frames in the animated image.

```
public int GetFrameCount()
```

Returns

[int](#)

The number of frames in the animated image.

## GetFrameInfo(int, out int, out int, out float)

Retrieves information of a specific frame in the animated image, including its dimensions and duration.

```
public void GetFrameInfo(int frameIndex, out int width, out int height, out float duration)
```

Parameters

**frameIndex** [int](#)

The index of the frame to retrieve information for.

**width** [int](#)

The output parameter that returns the width of the specified frame.

**height** [int](#)

The output parameter that returns the height of the specified frame.

**duration** [float](#)

The output parameter that returns the duration of the specified frame in milliseconds.

# Class Image

Namespace: [Bliss.CSharp.Images](#)

Assembly: Bliss.dll

```
public class Image : ICloneable
```

## Inheritance

[object](#) ← Image

## Implements

[ICloneable](#)

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### Image(byte[])

Initializes a new instance of the [Image](#) class from raw byte data.

```
public Image(byte[] data)
```

#### Parameters

**data** [byte](#)[]

The raw image data in RGBA format.

### Image(Stream)

Initializes a new instance of the [Image](#) class by loading an image from the specified stream.

```
public Image(Stream stream)
```

## Parameters

### `stream Stream`

The input stream containing image data to load.

## Exceptions

### `ArgumentException`

Thrown if the stream cannot be read.

## `Image(int, int, Color)`

Initializes a new instance of the `Image` class with a solid color.

```
public Image(int width, int height, Color color)
```

## Parameters

### `width int`

The width of the image in pixels.

### `height int`

The height of the image in pixels.

### `color Color`

The color to fill the image with.

## `Image(int, int, byte[]?)`

Initializes a new instance of the `Image` class with the specified dimensions and optional data.

```
public Image(int width, int height, byte[]? data = null)
```

## Parameters

**width** [int](#)

The width of the image in pixels.

**height** [int](#)

The height of the image in pixels.

**data** [byte](#)[]

The optional raw data to initialize the image with.

## Image(string)

Initializes a new instance of the [Image](#) class by loading an image from the specified file path.

```
public Image(string path)
```

### Parameters

**path** [string](#)

The file path of the image to load.

### Exceptions

[FileNotFoundException](#)

Thrown when the specified file does not exist.

## Properties

### Data

Gets or sets the raw image data as a byte array, typically in RGBA format.

```
public byte[] Data { get; set; }
```

### Property Value

[byte](#)[]

## Exceptions

[ArgumentException](#)

Thrown when the assigned data size does not match the product of the image's width, height, and the number of color components (4 for RGBA).

## Height

Gets or sets the height of the image in pixels.

```
public int Height { get; set; }
```

### Property Value

[int](#)

## Exceptions

[ArgumentOutOfRangeException](#)

Thrown when a non-positive value is assigned to the height.

## Width

Gets or sets the width of the image in pixels.

```
public int Width { get; set; }
```

### Property Value

[int](#)

## Exceptions

[ArgumentOutOfRangeException](#)

Thrown when a negative or zero value is assigned to the width.

## Methods

### Clone()

Creates a deep copy of this image.

```
public object Clone()
```

Returns

[object](#)

A new [Image](#) instance with identical data.

### Crop(Rectangle)

Crops the image to the specified rectangular region.

```
public void Crop(Rectangle crop)
```

Parameters

`crop` [Rectangle](#)

The rectangular area to crop the image to.

Exceptions

[ArgumentOutOfRangeException](#)

Thrown when the specified rectangle is outside the bounds of the image.

### FlipHorizontal()

Flips the image horizontally by mirroring its pixel data across the vertical axis.

```
public void FlipHorizontal()
```

## FlipVertical()

Flips the image vertically by inverting the order of rows in the image data.

```
public void FlipVertical()
```

## GetColor(int, int)

Gets the color of a specific pixel in the image.

```
public Color GetColor(int x, int y)
```

### Parameters

x [int](#)

The x-coordinate of the pixel.

y [int](#)

The y-coordinate of the pixel.

### Returns

[Color](#)

The color of the specified pixel.

## Resize(int, int)

Resizes the image to the specified width and height, maintaining proportional scaling.

```
public void Resize(int newWidth, int newHeight)
```

## Parameters

`newWidth` [int](#)

The desired width of the resized image.

`newHeight` [int](#)

The desired height of the resized image.

## ResizeNN(int, int)

Resizes the image to the specified width and height using the nearest-neighbor algorithm.

```
public void ResizeNN(int newWidth, int newHeight)
```

## Parameters

`newWidth` [int](#)

The new width of the image.

`newHeight` [int](#)

The new height of the image.

## Rotate(int)

Rotates the image by the specified degrees in a clockwise direction.

```
public void Rotate(int degrees)
```

## Parameters

`degrees` [int](#)

The angle in degrees to rotate the image. Must be a multiple of 90.

## RotateCCW()

Rotates the current [Image](#) instance counterclockwise by 90 degrees.

```
public void RotateCCW()
```

## RotateCW()

Rotates the current [Image](#) 90 degrees clockwise. Updates the image's dimensions and pixel data accordingly.

```
public void RotateCW()
```

## SaveAsBmp(string)

Saves the image as a BMP file.

```
public void SaveAsBmp(string path)
```

Parameters

**path** [string](#) ↗

The path where the BMP file should be saved.

## SaveAsHdr(string)

Saves the image as an HDR file.

```
public void SaveAsHdr(string path)
```

Parameters

**path** [string](#) ↗

The path where the HDR file should be saved.

## SaveAsJpg(string, int)

Saves the image as a JPG file with the specified quality.

```
public void SaveAsJpg(string path, int quality)
```

### Parameters

**path** [string](#)

The path where the JPG file should be saved.

**quality** [int](#)

The quality of the JPG image (1-100).

## SaveAsPng(string)

Saves the image as a PNG file.

```
public void SaveAsPng(string path)
```

### Parameters

**path** [string](#)

The path where the PNG file should be saved.

## SaveAsTga(string)

Saves the image as a TGA file.

```
public void SaveAsTga(string path)
```

### Parameters

**path** [string](#)

The path where the TGA file should be saved.

## SetPixel(int, int, Color)

Sets the color of a specific pixel in the image.

```
public void SetPixel(int x, int y, Color color)
```

### Parameters

x [int](#)

The x-coordinate of the pixel.

y [int](#)

The y-coordinate of the pixel.

color [Color](#)

The new color of the pixel.

## Tint(Color)

Applies a tint to the image by adjusting its pixel color values based on the specified tint color.

```
public void Tint(Color tint)
```

### Parameters

tint [Color](#)

The color tint to be applied to the image.

# Namespace Bliss.CSharp.Interact

## Classes

[Input](#)

# Class Input

Namespace: [Bliss.CSharp.Interact](#)

Assembly: Bliss.dll

```
public static class Input
```

## Inheritance

[object](#) ← Input

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Properties

## InputContext

Gets the current input context used to manage and handle input events.

```
public static IInputContext InputContext { get; }
```

## Property Value

[IInputContext](#)

# Methods

## Begin()

Begins input processing by interacting with the InputContext. This method should be called at the start of an input processing phase.

```
public static void Begin()
```

## Destroy()

Destroys the Input context and releases any associated resources.

```
public static void Destroy()
```

## DisableRelativeMouseMode()

Disables the relative mouse mode.

```
public static void DisableRelativeMouseMode()
```

## EnableRelativeMouseMode()

Enables relative mouse mode, which locks the cursor to the center of the window and provides relative motion data instead of absolute position.

```
public static void EnableRelativeMouseMode()
```

## End()

Finalizes the input handling for the current frame.

```
public static void End()
```

## GetAvailableGamepadCount()

Gets the count of available gamepads.

```
public static uint GetAvailableGamepadCount()
```

Returns

[uint](#)

The number of available gamepads.

## GetClipboardText()

Retrieves the current text from the system clipboard via the InputContext.

```
public static string GetClipboardText()
```

Returns

[string](#)

The current clipboard text.

## GetGamepadAxisMovement(uint, GamepadAxis)

Retrieves the movement value of the specified axis on the given gamepad.

```
public static float GetGamepadAxisMovement(uint gamepad, GamepadAxis axis)
```

Parameters

**gamepad** [uint](#)

The index of the gamepad.

**axis** [GamepadAxis](#)

The axis of the gamepad to check.

Returns

[float](#)

The movement value of the specified axis.

## GetGamepadName(uint)

Gets the name of the specified gamepad.

```
public static string GetGamepadName(uint gamepad)
```

## Parameters

**gamepad** [uint](#)

The index of the gamepad.

## Returns

[string](#)

The name of the specified gamepad.

## GetMouseCursor()

Gets the current mouse cursor from the input context.

```
public static ICursor GetMouseCursor()
```

## Returns

[ICursor](#)

## GetMouseDelta()

Retrieves the change in mouse position since the last frame.

```
public static Vector2 GetMouseDelta()
```

## Returns

[Vector2](#)

A Vector2 representing the delta of the mouse movement.

## GetMousePosition()

Gets the current position of the mouse in window coordinates.

```
public static Vector2 GetMousePosition()
```

Returns

[Vector2](#)

The current position of the mouse as a Vector2.

## GetPressedChars()

Retrieves characters that were pressed in the current frame.

```
public static char[] GetPressedChars()
```

Returns

[char](#)[]

An array of characters pressed in the current frame.

## HideCursor()

Hides the mouse cursor.

```
public static void HideCursor()
```

## Init(IInputContext)

Initializes the Input class with the specified input context.

```
public static void Init(IInputContext inputContext)
```

## Parameters

**inputContext** [IInputContext](#)

The input context to initialize.

## IsCursorShown()

Checks if the cursor is currently shown.

```
public static bool IsCursorShown()
```

## Returns

[bool](#)

True if the cursor is shown; otherwise, false.

## IsFileDragDropped(out string)

Determines whether a file has been drag-dropped onto the application.

```
public static bool IsFileDragDropped(out string path)
```

## Parameters

**path** [string](#)

The path of the drag-dropped file, if one exists.

## Returns

[bool](#)

True if a file was drag-dropped; otherwise, false.

## IsGamepadAvailable(uint)

Checks if the specified gamepad is available.

```
public static bool IsGamepadAvailable(uint gamepad)
```

Parameters

gamepad [uint](#)

The index of the gamepad to check.

Returns

[bool](#)

True if the gamepad is available; otherwise, false.

## IsGamepadButtonDown(uint, GamepadButton)

Checks if a specific gamepad button is currently pressed.

```
public static bool IsGamepadButtonDown(uint gamepad, GamepadButton button)
```

Parameters

gamepad [uint](#)

The ID of the gamepad to check.

button [GamepadButton](#)

The gamepad button to check.

Returns

[bool](#)

True if the button is pressed, otherwise false.

## IsGamepadButtonPressed(uint, GamepadButton)

Checks if the specified button on the given gamepad is pressed.

```
public static bool IsGamepadButtonPressed(uint gamepad, GamepadButton button)
```

Parameters

**gamepad** [uint](#)

The identifier of the gamepad to check.

**button** [GamepadButton](#)

The button on the gamepad to check.

Returns

[bool](#)

True if the button is pressed; otherwise, false.

## IsGamepadButtonReleased(uint, GamepadButton)

Checks if the specified gamepad button is released.

```
public static bool IsGamepadButtonReleased(uint gamepad, GamepadButton button)
```

Parameters

**gamepad** [uint](#)

The gamepad identifier.

**button** [GamepadButton](#)

The gamepad button to check.

Returns

[bool](#)

True if the specified gamepad button is released; otherwise, false.

## IsGamepadButtonUp(uint, GamepadButton)

Checks if the specified gamepad button is not pressed.

```
public static bool IsGamepadButtonUp(uint gamepad, GamepadButton button)
```

Parameters

**gamepad** [uint](#)

The identifier of the gamepad.

**button** [GamepadButton](#)

The gamepad button to check.

Returns

[bool](#)

True if the specified gamepad button is up; otherwise, false.

## IsKeyDown(KeyboardKey)

Checks if the specified keyboard key is currently pressed down.

```
public static bool IsKeyDown(KeyboardKey key)
```

Parameters

**key** [KeyboardKey](#)

The keyboard key to check.

Returns

[bool](#)

True if the key is pressed down; otherwise, false.

## IsKeyPressed(KeyboardKey)

Checks if a specific key on the keyboard is currently pressed.

```
public static bool IsKeyPressed(KeyboardKey key)
```

Parameters

**key** [KeyboardKey](#)

The keyboard key to check.

Returns

[bool](#)

## IsKeyReleased(KeyboardKey)

Checks if the specified key has been released.

```
public static bool IsKeyReleased(KeyboardKey key)
```

Parameters

**key** [KeyboardKey](#)

The keyboard key to check.

Returns

[bool](#)

True if the key has been released; otherwise, false.

## IsKeyUp(KeyboardKey)

Determines whether the specified key is currently up (not pressed).

```
public static bool IsKeyUp(KeyboardKey key)
```

## Parameters

### key [KeyboardKey](#)

The keyboard key to check.

## Returns

### [bool](#)

True if the key is up; otherwise, false.

## IsMouseButtonDown(MouseButton)

Checks if the specified mouse button is currently being pressed down.

```
public static bool IsMouseButtonDown(MouseButton button)
```

## Parameters

### button [MouseButton](#)

The mouse button to check.

## Returns

### [bool](#)

True if the button is down; otherwise, false.

## IsMouseButtonPressed(MouseButton)

Checks if the specified mouse button was pressed in the current frame.

```
public static bool IsMouseButtonPressed(MouseButton button)
```

## Parameters

**button** [MouseButton](#)

The mouse button to check.

## Returns

[bool](#) ↗

True if the button was pressed; otherwise, false.

## IsMouseButtonReleased(MouseButton)

Checks if the specified mouse button was released in the current frame.

```
public static bool IsMouseButtonReleased(MouseButton button)
```

## Parameters

**button** [MouseButton](#)

The mouse button to check.

## Returns

[bool](#) ↗

True if the button was released; otherwise, false.

## IsMouseButtonUp(MouseButton)

Checks if the specified mouse button is currently up (not pressed).

```
public static bool IsMouseButtonUp(MouseButton button)
```

## Parameters

**button** [MouseButton](#)

The mouse button to check.

Returns

[bool](#)

True if the button is up; otherwise, false.

## IsMouseMoving(out Vector2)

Checks if the mouse is currently moving and provides its position.

```
public static bool IsMouseMoving(out Vector2 position)
```

Parameters

[position](#) [Vector2](#)

The current position of the mouse if it is moving.

Returns

[bool](#)

True if the mouse is moving; otherwise, false.

## IsMouseScrolling(out Vector2)

Checks if the mouse is currently scrolling and retrieves the wheel delta if it is.

```
public static bool IsMouseScrolling(out Vector2 wheelDelta)
```

Parameters

[wheelDelta](#) [Vector2](#)

The vector representing the scroll delta of the mouse.

Returns

[bool](#)

True if the mouse is scrolling, otherwise false.

## IsRelativeMouseModeEnabled()

Checks if relative mouse mode is enabled, where the cursor is locked to the window.

```
public static bool IsRelativeMouseModeEnabled()
```

Returns

[bool](#)

True if relative mouse mode is enabled; otherwise, false.

## RumbleGamepad(uint, ushort, ushort, uint)

Generates a rumble effect on the specified gamepad.

```
public static void RumbleGamepad(uint gamepad, ushort lowFrequencyRumble, ushort  
highFrequencyRumble, uint durationMs)
```

Parameters

**gamepad** [uint](#)

The index of the gamepad to rumble.

**lowFrequencyRumble** [ushort](#)

The intensity of the low-frequency rumble.

**highFrequencyRumble** [ushort](#)

The intensity of the high-frequency rumble.

**durationMs** [uint](#)

Duration of the rumble effect in milliseconds.

## SetClipboardText(string)

Sets the clipboard content to the specified text.

```
public static void SetClipboardText(string text)
```

Parameters

**text** [string](#)

The text to set to the clipboard.

## SetMouseCursor(ICursor)

Sets the mouse cursor to the specified cursor.

```
public static void SetMouseCursor(ICursor cursor)
```

Parameters

**cursor** [ICursor](#)

The cursor to set.

## SetMousePosition(Vector2)

Sets the mouse position to the specified coordinates.

```
public static void SetMousePosition(Vector2 position)
```

Parameters

**position** [Vector2](#)

The position to set the mouse to.

## ShowCursor()

Shows the mouse cursor.

```
public static void ShowCursor()
```

# Namespace Bliss.CSharp.Interact.Contexts

## Classes

[Sdl3InputContext](#)

## Interfaces

[IInputContext](#)

# Interface IInputContext

Namespace: [Bliss.CSharp.Interact.Contexts](#)

Assembly: Bliss.dll

```
public interface IInputContext : IDisposable
```

## Inherited Members

[IDisposable.Dispose\(\)](#) ↗

## Methods

### Begin()

Begins input processing for the current frame.

```
void Begin()
```

### DisableRelativeMouseMode()

Disables relative mouse mode.

```
void DisableRelativeMouseMode()
```

### EnableRelativeMouseMode()

Enables relative mouse mode.

```
void EnableRelativeMouseMode()
```

### End()

Ends input processing for the current frame.

```
void End()
```

## GetAvailableGamepadCount()

Gets the count of available gamepads.

```
uint GetAvailableGamepadCount()
```

Returns

[uint](#)

The number of available gamepads.

## GetClipboardText()

Retrieves the current text from the system clipboard.

```
string GetClipboardText()
```

Returns

[string](#)

The clipboard text.

## GetGamepadAxisMovement(uint, GamepadAxis)

Retrieves the movement value of the specified axis on the given gamepad.

```
float GetGamepadAxisMovement(uint gamepad, GamepadAxis axis)
```

Parameters

gamepad [uint](#)

The index of the gamepad.

#### **axis** [GamepadAxis](#)

The axis of the gamepad to check.

Returns

#### [float](#)

The movement value of the specified axis.

## GetGamepadName(uint)

Gets the name of the specified gamepad.

[string](#) [GetGamepadName](#)([uint](#) gamepad)

Parameters

#### **gamepad** [uint](#)

The index of the gamepad.

Returns

#### [string](#)

The name of the specified gamepad.

## GetMouseCursor()

Gets the current mouse cursor.

[ICursor](#) [GetMouseCursor](#)()

Returns

#### [ICursor](#)

The current mouse cursor.

## GetMouseDelta()

Retrieves the change in mouse position since the last frame.

`Vector2 GetMouseDelta()`

Returns

[Vector2](#)

A `Vector2` representing the delta of the mouse movement.

## GetMousePosition()

Gets the current position of the mouse in window coordinates.

`Vector2 GetMousePosition()`

Returns

[Vector2](#)

The mouse position as a [Vector2](#).

## GetPressedChars()

Retrieves characters that were pressed in the current frame.

`char[] GetPressedChars()`

Returns

[char](#)[]

An array of characters pressed in the current frame.

## HideCursor()

Hides the mouse cursor.

```
void HideCursor()
```

## IsCursorShown()

Checks if the cursor is currently shown.

```
bool IsCursorShown()
```

Returns

[bool](#)

True if the cursor is shown; otherwise, false.

## IsFileDragDropped(out string)

Checks if a file has been drag-dropped onto the application.

```
bool IsFileDragDropped(out string path)
```

Parameters

[path](#) [string](#)

When this method returns, contains the path of the file that was drag-dropped.

Returns

[bool](#)

True if a file was drag-dropped; otherwise, false.

## IsGamepadAvailable(uint)

Checks if the specified gamepad is available.

```
bool IsGamepadAvailable(uint gamepad)
```

Parameters

gamepad [uint](#)

The index of the gamepad to check.

Returns

[bool](#)

True if the gamepad is available; otherwise, false.

## IsGamepadButtonDown(uint, GamepadButton)

Checks if the specified button on the given gamepad is currently being pressed.

```
bool IsGamepadButtonDown(uint gamepad, GamepadButton button)
```

Parameters

gamepad [uint](#)

The index of the gamepad.

button [GamepadButton](#)

The button on the gamepad to check.

Returns

[bool](#)

True if the specified button is pressed, otherwise false.

## IsGamepadButtonPressed(uint, GamepadButton)

Checks if a specific gamepad button is pressed.

```
bool IsGamepadButtonPressed(uint gamepad, GamepadButton button)
```

Parameters

gamepad [uint](#)

The identifier of the gamepad.

button [GamepadButton](#)

The button on the gamepad to check.

Returns

[bool](#)

True if the specified button is pressed; otherwise, false.

## IsGamepadButtonReleased(uint, GamepadButton)

Determines whether a specific button on a specified gamepad has been released.

```
bool IsGamepadButtonReleased(uint gamepad, GamepadButton button)
```

Parameters

gamepad [uint](#)

The identifier of the gamepad.

button [GamepadButton](#)

The gamepad button to check.

Returns

[bool](#)

True if the button was released; otherwise, false.

## IsGamepadButtonUp(uint, GamepadButton)

Determines whether a specified button on the specified gamepad is currently not pressed.

```
bool IsGamepadButtonUp(uint gamepad, GamepadButton button)
```

### Parameters

**gamepad** [uint](#)

The identifier of the gamepad.

**button** [GamepadButton](#)

The gamepad button to check.

### Returns

[bool](#)

True if the button is up (not pressed); otherwise, false.

## IsKeyDown(KeyboardKey)

Checks if a specified key is currently pressed down.

```
bool IsKeyDown(KeyboardKey key)
```

### Parameters

**key** [KeyboardKey](#)

The key to check the state of.

### Returns

[bool](#)

True if the key is pressed, false otherwise.

## IsKeyPressed(KeyboardKey)

Checks if the specified keyboard key is currently pressed.

```
bool IsKeyPressed(KeyboardKey key)
```

Parameters

**key** [KeyboardKey](#)

The keyboard key to check.

Returns

[bool](#)

True if the specified key is pressed; otherwise, false.

## IsKeyReleased(KeyboardKey)

Checks if a specified keyboard key has been released.

```
bool IsKeyReleased(KeyboardKey key)
```

Parameters

**key** [KeyboardKey](#)

The keyboard key to check.

Returns

[bool](#)

True if the key has been released, otherwise false.

## IsKeyUp(KeyboardKey)

Checks if a specific keyboard key is currently in the released state.

```
bool IsKeyUp(KeyboardKey key)
```

## Parameters

**key** [KeyboardKey](#)

The keyboard key to check.

## Returns

[bool](#)

Returns true if the specified key is up; otherwise, false.

## IsMouseButtonDown(MouseButton)

Checks if the specified mouse button is currently being held down.

```
bool IsMouseButtonDown(MouseButton button)
```

## Parameters

**button** [MouseButton](#)

The mouse button to check.

## Returns

[bool](#)

True if the button is down; otherwise, false.

## IsMouseButtonPressed(MouseButton)

Checks if the specified mouse button was pressed in the current frame.

```
bool IsMouseButtonPressed(MouseButton button)
```

## Parameters

**button** [MouseButton](#)

The mouse button to check.

## Returns

[bool](#) ↗

True if the button was pressed; otherwise, false.

## IsMouseButtonReleased(MouseButton)

Checks if the specified mouse button was released in the current frame.

**bool** [IsMouseButtonReleased](#)(MouseButton button)

## Parameters

**button** [MouseButton](#)

The mouse button to check.

## Returns

[bool](#) ↗

True if the button was released; otherwise, false.

## IsMouseButtonUp(MouseButton)

Checks if the specified mouse button is currently up (not pressed).

**bool** [IsMouseButtonUp](#)(MouseButton button)

## Parameters

**button** [MouseButton](#)

The mouse button to check.

Returns

[bool](#)

True if the button is up; otherwise, false.

## IsMouseMoving(out Vector2)

Checks if the mouse is moving and returns the current position if it is.

```
bool IsMouseMoving(out Vector2 position)
```

Parameters

[position](#) [Vector2](#)

The current mouse position.

Returns

[bool](#)

True if the mouse is moving; otherwise, false.

## IsMouseScrolling(out Vector2)

Checks if the mouse is scrolling and returns the scroll delta if it is.

```
bool IsMouseScrolling(out Vector2 wheelDelta)
```

Parameters

[wheelDelta](#) [Vector2](#)

The scroll delta of the mouse.

Returns

[bool](#)

True if the mouse is scrolling; otherwise, false.

## IsRelativeMouseModeEnabled()

Checks if relative mouse mode is enabled, where the cursor is locked to the window.

`bool IsRelativeMouseModeEnabled()`

Returns

[bool](#)

True if relative mouse mode is enabled; otherwise, false.

## RumbleGamepad(uint, ushort, ushort, uint)

Generates a rumble effect on the specified gamepad.

`void RumbleGamepad(uint gamepad, ushort lowFrequencyRumble, ushort highFrequencyRumble, uint durationMs)`

Parameters

`gamepad` [uint](#)

The index of the gamepad to rumble.

`lowFrequencyRumble` [ushort](#)

The intensity of the low-frequency rumble.

`highFrequencyRumble` [ushort](#)

The intensity of the high-frequency rumble.

`durationMs` [uint](#)

Duration of the rumble effect in milliseconds.

## SetClipboardText(string)

Sets the clipboard content to the specified text.

```
void SetClipboardText(string text)
```

### Parameters

**text** [string](#)

The text to set to the clipboard.

## SetMouseCursor(ICursor)

Sets the mouse cursor to the specified cursor.

```
void SetMouseCursor(ICursor cursor)
```

### Parameters

**cursor** [ICursor](#)

The cursor to set.

## SetMousePosition(Vector2)

Sets the mouse position to the specified coordinates.

```
void SetMousePosition(Vector2 position)
```

### Parameters

**position** [Vector2](#)

The position to set the mouse to.

## ShowCursor()

Shows the mouse cursor.

```
void ShowCursor()
```

# Class Sdl3InputContext

Namespace: [Bliss.CSharp.Interact.Contexts](#)

Assembly: Bliss.dll

```
public class Sdl3InputContext : Disposable, IInputContext, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← Sdl3InputContext

## Implements

[IInputContext](#), [IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### Sdl3InputContext(IWindow)

Initializes a new instance of the [Sdl3InputContext](#) class for handling input events from a window.

```
public Sdl3InputContext(IWindow window)
```

## Parameters

window [IWindow](#)

The window associated with the input context. Must be an SDL3 window.

## Exceptions

[Exception](#)

Thrown if the provided window is not an SDL3 window.

# Methods

## Begin()

Begins input processing for the current frame.

```
public void Begin()
```

## DisableRelativeMouseMode()

Disables relative mouse mode.

```
public void DisableRelativeMouseMode()
```

## Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

Parameters

`disposing` [bool](#)

True if called from user code; false if called from a finalizer.

## EnableRelativeMouseMode()

Enables relative mouse mode.

```
public void EnableRelativeMouseMode()
```

## End()

Ends input processing for the current frame.

```
public void End()
```

## GetAvailableGamepadCount()

Gets the count of available gamepads.

```
public uint GetAvailableGamepadCount()
```

Returns

[uint](#)

The number of available gamepads.

## GetClipboardText()

Retrieves the current text from the system clipboard.

```
public string GetClipboardText()
```

Returns

[string](#)

The clipboard text.

## GetGamepadAxisMovement(uint, GamepadAxis)

Retrieves the movement value of the specified axis on the given gamepad.

```
public float GetGamepadAxisMovement(uint gamepad, GamepadAxis axis)
```

Parameters

gamepad [uint](#)

The index of the gamepad.

#### **axis** [GamepadAxis](#)

The axis of the gamepad to check.

Returns

#### [float](#)

The movement value of the specified axis.

## GetGamepadName(uint)

Gets the name of the specified gamepad.

```
public string GetGamepadName(uint gamepad)
```

Parameters

#### **gamepad** [uint](#)

The index of the gamepad.

Returns

#### [string](#)

The name of the specified gamepad.

## GetMouseCursor()

Gets the current mouse cursor.

```
public ICursor GetMouseCursor()
```

Returns

#### [ICursor](#)

The current mouse cursor.

## GetMouseDelta()

Retrieves the change in mouse position since the last frame.

```
public Vector2 GetMouseDelta()
```

Returns

[Vector2](#)

A Vector2 representing the delta of the mouse movement.

## GetMousePosition()

Gets the current position of the mouse in window coordinates.

```
public Vector2 GetMousePosition()
```

Returns

[Vector2](#)

The mouse position as a [Vector2](#).

## GetPressedChars()

Retrieves characters that were pressed in the current frame.

```
public char[] GetPressedChars()
```

Returns

[char](#)[]

An array of characters pressed in the current frame.

## HideCursor()

Hides the mouse cursor.

```
public void HideCursor()
```

## IsCursorShown()

Checks if the cursor is currently shown.

```
public bool IsCursorShown()
```

Returns

[bool](#)

True if the cursor is shown; otherwise, false.

## IsFileDragDropped(out string)

Checks if a file has been drag-dropped onto the application.

```
public bool IsFileDragDropped(out string path)
```

Parameters

[path](#) [string](#)

When this method returns, contains the path of the file that was drag-dropped.

Returns

[bool](#)

True if a file was drag-dropped; otherwise, false.

## IsGamepadAvailable(uint)

Checks if the specified gamepad is available.

```
public bool IsGamepadAvailable(uint gamepad)
```

Parameters

**gamepad** [uint](#)

The index of the gamepad to check.

Returns

[bool](#)

True if the gamepad is available; otherwise, false.

## IsGamepadButtonDown(uint, GamepadButton)

Checks if the specified button on the given gamepad is currently being pressed.

```
public bool IsGamepadButtonDown(uint gamepad, GamepadButton button)
```

Parameters

**gamepad** [uint](#)

The index of the gamepad.

**button** [GamepadButton](#)

The button on the gamepad to check.

Returns

[bool](#)

True if the specified button is pressed, otherwise false.

## IsGamepadButtonPressed(uint, GamepadButton)

Checks if a specific gamepad button is pressed.

```
public bool IsGamepadButtonPressed(uint gamepad, GamepadButton button)
```

Parameters

**gamepad** [uint](#)

The identifier of the gamepad.

**button** [GamepadButton](#)

The button on the gamepad to check.

Returns

[bool](#)

True if the specified button is pressed; otherwise, false.

## IsGamepadButtonReleased(uint, GamepadButton)

Determines whether a specific button on a specified gamepad has been released.

```
public bool IsGamepadButtonReleased(uint gamepad, GamepadButton button)
```

Parameters

**gamepad** [uint](#)

The identifier of the gamepad.

**button** [GamepadButton](#)

The gamepad button to check.

Returns

[bool](#)

True if the button was released; otherwise, false.

## IsGamepadButtonUp(uint, GamepadButton)

Determines whether a specified button on the specified gamepad is currently not pressed.

```
public bool IsGamepadButtonUp(uint gamepad, GamepadButton button)
```

### Parameters

**gamepad** [uint](#)

The identifier of the gamepad.

**button** [GamepadButton](#)

The gamepad button to check.

### Returns

[bool](#)

True if the button is up (not pressed); otherwise, false.

## IsKeyDown(KeyboardKey)

Checks if a specified key is currently pressed down.

```
public bool IsKeyDown(KeyboardKey key)
```

### Parameters

**key** [KeyboardKey](#)

The key to check the state of.

### Returns

[bool](#)

True if the key is pressed, false otherwise.

## IsKeyPressed(KeyboardKey)

Checks if the specified keyboard key is currently pressed.

```
public bool IsKeyPressed(KeyboardKey key)
```

Parameters

**key** [KeyboardKey](#)

The keyboard key to check.

Returns

[bool](#)

True if the specified key is pressed; otherwise, false.

## IsKeyReleased(KeyboardKey)

Checks if a specified keyboard key has been released.

```
public bool IsKeyReleased(KeyboardKey key)
```

Parameters

**key** [KeyboardKey](#)

The keyboard key to check.

Returns

[bool](#)

True if the key has been released, otherwise false.

## IsKeyUp(KeyboardKey)

Checks if a specific keyboard key is currently in the released state.

```
public bool IsKeyUp(KeyboardKey key)
```

## Parameters

### key [KeyboardKey](#)

The keyboard key to check.

## Returns

### [bool](#)

Returns true if the specified key is up; otherwise, false.

## IsMouseButtonDown(MouseButton)

Checks if the specified mouse button is currently being held down.

```
public bool IsMouseButtonDown(MouseButton button)
```

## Parameters

### button [MouseButton](#)

The mouse button to check.

## Returns

### [bool](#)

True if the button is down; otherwise, false.

## IsMouseButtonPressed(MouseButton)

Checks if the specified mouse button was pressed in the current frame.

```
public bool IsMouseButtonPressed(MouseButton button)
```

## Parameters

**button** [MouseButton](#)

The mouse button to check.

## Returns

[bool](#) ↗

True if the button was pressed; otherwise, false.

## IsMouseButtonReleased(MouseButton)

Checks if the specified mouse button was released in the current frame.

```
public bool IsMouseButtonReleased(MouseButton button)
```

## Parameters

**button** [MouseButton](#)

The mouse button to check.

## Returns

[bool](#) ↗

True if the button was released; otherwise, false.

## IsMouseButtonUp(MouseButton)

Checks if the specified mouse button is currently up (not pressed).

```
public bool IsMouseButtonUp(MouseButton button)
```

## Parameters

**button** [MouseButton](#)

The mouse button to check.

Returns

[bool](#)

True if the button is up; otherwise, false.

## IsMouseMoving(out Vector2)

Checks if the mouse is moving and returns the current position if it is.

```
public bool IsMouseMoving(out Vector2 position)
```

Parameters

[position](#) [Vector2](#)

The current mouse position.

Returns

[bool](#)

True if the mouse is moving; otherwise, false.

## IsMouseScrolling(out Vector2)

Checks if the mouse is scrolling and returns the scroll delta if it is.

```
public bool IsMouseScrolling(out Vector2 wheelDelta)
```

Parameters

[wheelDelta](#) [Vector2](#)

The scroll delta of the mouse.

Returns

[bool](#)

True if the mouse is scrolling; otherwise, false.

## IsRelativeMouseModeEnabled()

Checks if relative mouse mode is enabled, where the cursor is locked to the window.

```
public bool IsRelativeMouseModeEnabled()
```

Returns

[bool](#)

True if relative mouse mode is enabled; otherwise, false.

## RumbleGamepad(uint, ushort, ushort, uint)

Generates a rumble effect on the specified gamepad.

```
public void RumbleGamepad(uint gamepad, ushort lowFrequencyRumble, ushort  
highFrequencyRumble, uint durationMs)
```

Parameters

**gamepad** [uint](#)

The index of the gamepad to rumble.

**lowFrequencyRumble** [ushort](#)

The intensity of the low-frequency rumble.

**highFrequencyRumble** [ushort](#)

The intensity of the high-frequency rumble.

**durationMs** [uint](#)

Duration of the rumble effect in milliseconds.

## SetClipboardText(string)

Sets the clipboard content to the specified text.

```
public void SetClipboardText(string text)
```

Parameters

**text** [string](#)

The text to set to the clipboard.

## SetMouseCursor(ICursor)

Sets the mouse cursor to the specified cursor.

```
public void SetMouseCursor(ICursor cursor)
```

Parameters

**cursor** [ICursor](#)

The cursor to set.

## SetMousePosition(Vector2)

Sets the mouse position to the specified coordinates.

```
public void SetMousePosition(Vector2 position)
```

Parameters

**position** [Vector2](#)

The position to set the mouse to.

## ShowCursor()

Shows the mouse cursor.

```
public void ShowCursor()
```

# Namespace Bliss.CSharp.Interact.Gamepads

## Classes

[Sdl3Gamepad](#)

## Interfaces

[IGamepad](#)

## Enums

[GamepadAxis](#)

[GamepadButton](#)

# Enum GamepadAxis

Namespace: [Bliss.CSharp.Interact.Gamepads](#)

Assembly: Bliss.dll

```
public enum GamepadAxis
```

## Fields

**Invalid** = -1

Invalid axis.

**LeftX** = 0

Left stick horizontal axis.

**LeftY** = 1

Left stick vertical axis.

**Max** = 6

Maximum axis value.

**RightX** = 2

Right stick horizontal axis.

**RightY** = 3

Right stick vertical axis.

**TriggerLeft** = 4

Left trigger axis.

**TriggerRight** = 5

Right trigger axis.

# Enum GamepadButton

Namespace: [Bliss.CSharp.Interact.Gamepads](#)

Assembly: Bliss.dll

```
public enum GamepadButton
```

## Fields

**Back** = 4

The "Back" button on the gamepad.

**Count** = 26

Represents the total number of available buttons.

**DpadDown** = 12

The directional pad "Down" button.

**DpadLeft** = 13

The directional pad "Left" button.

**DpadRight** = 14

The directional pad "Right" button.

**DpadUp** = 11

The directional pad "Up" button.

**East** = 1

The "East" face button (often the right button, typically labeled B or Circle).

**Guide** = 5

The "Guide" button, often used to open system menus.

**Invalid** = -1

Represents an invalid button.

**LeftPaddle1 = 17**

The first left paddle button.

**LeftPaddle2 = 19**

The second left paddle button.

**LeftShoulder = 9**

The left shoulder button.

**LeftStick = 7**

The left analog stick button (when pressed down).

**Misc1 = 15**

A miscellaneous button (Misc1).

**Misc2 = 21**

A miscellaneous button (Misc2).

**Misc3 = 22**

A miscellaneous button (Misc3).

**Misc4 = 23**

A miscellaneous button (Misc4).

**Misc5 = 24**

A miscellaneous button (Misc5).

**Misc6 = 25**

A miscellaneous button (Misc6).

**North = 3**

The "North" face button (often the top button, typically labeled Y or Triangle).

**RightPaddle1 = 16**

The first right paddle button.

**RightPaddle2** = 18

The second right paddle button.

**RightShoulder** = 10

The right shoulder button.

**RightStick** = 8

The right analog stick button (when pressed down).

**South** = 0

The "South" face button (often the bottom button, typically labeled A or X).

**Start** = 6

The "Start" button on the gamepad.

**Touchpad** = 20

The touchpad button, commonly found on some controllers.

**West** = 2

The "West" face button (often the left button, typically labeled X or Square).

# Interface IGamepad

Namespace: [Bliss.CSharp.Interact.Gamepads](#)

Assembly: Bliss.dll

```
public interface IGamepad : IDisposable
```

## Inherited Members

[IDisposable.Dispose\(\)](#)

## Methods

### CleanStates()

Cleans or resets the internal states of the gamepad (e.g., button states).

```
void CleanStates()
```

### GetAxisMovement(GamepadAxis)

Gets the movement value of a specified axis on the gamepad.

```
float GetAxisMovement(GamepadAxis axis)
```

#### Parameters

**axis** [GamepadAxis](#)

The axis to check.

#### Returns

[float](#)

A float representing the axis movement.

## GetHandle()

Gets the handle of the gamepad.

```
nint GetHandle()
```

Returns

[nint](#)

An integer pointer representing the gamepad's handle.

## GetIndex()

Gets the index of the gamepad.

```
uint GetIndex()
```

Returns

[uint](#)

An unsigned integer representing the gamepad's index.

## GetName()

Gets the name of the gamepad.

```
string GetName()
```

Returns

[string](#)

A string representing the gamepad's name.

## IsButtonDown(GamepadButton)

Checks if the specified button is currently being held down.

```
bool IsButtonDown(GamepadButton button)
```

Parameters

**button** [GamepadButton](#)

The button to check.

Returns

[bool](#)

True if the button is down; otherwise, false.

## IsButtonPressed(GamepadButton)

Checks if the specified button was pressed in the current frame.

```
bool IsButtonPressed(GamepadButton button)
```

Parameters

**button** [GamepadButton](#)

The button to check.

Returns

[bool](#)

True if the button was pressed; otherwise, false.

## IsButtonReleased(GamepadButton)

Checks if the specified button was released in the current frame.

```
bool IsButtonReleased(GamepadButton button)
```

Parameters

**button** [GamepadButton](#)

The button to check.

Returns

[bool](#) ↗

True if the button was released; otherwise, false.

## IsButtonUp(GamepadButton)

Checks if the specified button is currently up (not pressed).

```
bool IsButtonUp(GamepadButton button)
```

Parameters

**button** [GamepadButton](#)

The button to check.

Returns

[bool](#) ↗

True if the button is up; otherwise, false.

# Class Sdl3Gamepad

Namespace: [Bliss.CSharp.Interact.Gamepads](#)

Assembly: Bliss.dll

```
public class Sdl3Gamepad : Disposable, IGamepad, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← Sdl3Gamepad

## Implements

[IGamepad](#), [IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### Sdl3Gamepad(IWindow, uint)

Initializes a new instance of the [Sdl3Gamepad](#) class, which manages gamepad input for a specific window.

```
public Sdl3Gamepad(IWindow window, uint index)
```

## Parameters

**window** [IWindow](#)

The window associated with the gamepad.

**index** [uint](#)

The index of the gamepad to be opened.

# Properties

## Window

Gets the window associated with the gamepad.

```
public IWindow Window { get; }
```

## Property Value

[IWindow](#)

# Methods

## CleanStates()

Cleans or resets the internal states of the gamepad (e.g., button states).

```
public void CleanStates()
```

## Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

## Parameters

**disposing** [bool](#)

True if called from user code; false if called from a finalizer.

## GetAxisMovement(GamepadAxis)

Gets the movement value of a specified axis on the gamepad.

```
public float GetAxisMovement(GamepadAxis axis)
```

Parameters

**axis** [GamepadAxis](#)

The axis to check.

Returns

[float](#)

A float representing the axis movement.

## GetHandle()

Gets the handle of the gamepad.

```
public nint GetHandle()
```

Returns

[nint](#)

An integer pointer representing the gamepad's handle.

## GetIndex()

Gets the index of the gamepad.

```
public uint GetIndex()
```

Returns

[uint](#)

An unsigned integer representing the gamepad's index.

## GetName()

Gets the name of the gamepad.

```
public string GetName()
```

Returns

[string](#)

A string representing the gamepad's name.

## IsButtonDown(GamepadButton)

Checks if the specified button is currently being held down.

```
public bool IsButtonDown(GamepadButton button)
```

Parameters

[button](#) [GamepadButton](#)

The button to check.

Returns

[bool](#)

True if the button is down; otherwise, false.

## IsButtonPressed(GamepadButton)

Checks if the specified button was pressed in the current frame.

```
public bool IsButtonPressed(GamepadButton button)
```

Parameters

## `button` [GamepadButton](#)

The button to check.

Returns

### [bool](#) ↗

True if the button was pressed; otherwise, false.

## `IsButtonReleased(GamepadButton)`

Checks if the specified button was released in the current frame.

```
public bool IsButtonReleased(GamepadButton button)
```

Parameters

### `button` [GamepadButton](#)

The button to check.

Returns

### [bool](#) ↗

True if the button was released; otherwise, false.

## `IsButtonUp(GamepadButton)`

Checks if the specified button is currently up (not pressed).

```
public bool IsButtonUp(GamepadButton button)
```

Parameters

### `button` [GamepadButton](#)

The button to check.

Returns

bool ↗

True if the button is up; otherwise, false.

# Namespace Bliss.CSharp.Interact.Keyboards

## Enums

[KeyboardKey](#)

# Enum KeyboardKey

Namespace: [Bliss.CSharp.Interact.Keyboards](#)

Assembly: Bliss.dll

```
public enum KeyboardKey
```

## Fields

**A = 83**

Letter key A.

**AltLeft = 5**

The left Alt key.

**AltRight = 6**

The right Alt key.

**B = 84**

Letter key B.

**BackSlash = 129**

The Backslash key.

**BackSpace = 53**

The Backspace key.

**BracketLeft = 122**

The left bracket key.

**BracketRight = 123**

The right bracket key.

**C = 85**

Letter key C.

**CapsLock** = 60

The Caps Lock key.

**Clear** = 65

The Clear key.

**Comma** = 126

The Comma key.

**ControlLeft** = 3

The left Control key.

**ControlRight** = 4

The right Control key.

**D** = 86

Letter key D.

**Delete** = 55

The Delete key.

**Down** = 46

The Down arrow key.

**E** = 87

Letter key E.

**End** = 59

The End key.

**Enter** = 49

The Enter key.

**Escape** = 50

The Escape key.

**F = 88**

Letter key F.

**F1 = 10**

Function key F1.

**F10 = 19**

Function key F10.

**F11 = 20**

Function key F11.

**F12 = 21**

Function key F12.

**F13 = 22**

Function key F13.

**F14 = 23**

Function key F14.

**F15 = 24**

Function key F15.

**F16 = 25**

Function key F16.

**F17 = 26**

Function key F17.

**F18 = 27**

Function key F18.

**F19 = 28**

Function key F19.

**F2 = 11**

Function key F2.

**F20 = 29**

Function key F20.

**F21 = 30**

Function key F21.

**F22 = 31**

Function key F22.

**F23 = 32**

Function key F23.

**F24 = 33**

Function key F24.

**F25 = 34**

Function key F25.

**F26 = 35**

Function key F26.

**F27 = 36**

Function key F27.

**F28 = 37**

Function key F28.

**F29 = 38**

Function key F29.

**F3 = 12**

Function key F3.

**F30 = 39**

Function key F30.

**F31 = 40**

Function key F31.

**F32 = 41**

Function key F32.

**F33 = 42**

Function key F33.

**F34 = 43**

Function key F34.

**F35 = 44**

Function key F35.

**F4 = 13**

Function key F4.

**F5 = 14**

Function key F5.

**F6 = 15**

Function key F6.

**F7 = 16**

Function key F7.

**F8 = 17**

Function key F8.

**F9 = 18**

Function key F9.

**G = 89**

Letter key G.

**Grave = 119**

The Grave key, also known as the Tilde key.

**H = 90**

Letter key H.

**Home = 58**

The Home key.

**I = 91**

Letter key I.

**Insert = 54**

The Insert key.

**J = 92**

Letter key J.

**K = 93**

Letter key K.

**Keypad0 = 67**

Numeric keypad key 0.

**Keypad1 = 68**

Numeric keypad key 1.

**Keypad2 = 69**

Numeric keypad key 2.

**Keypad3 = 70**

Numeric keypad key 3.

**Keypad4** = 71

Numeric keypad key 4.

**Keypad5** = 72

Numeric keypad key 5.

**Keypad6** = 73

Numeric keypad key 6.

**Keypad7** = 74

Numeric keypad key 7.

**Keypad8** = 75

Numeric keypad key 8.

**Keypad9** = 76

Numeric keypad key 9.

**KeypadDecimal** = 81

Numeric keypad decimal point key.

**KeypadDivide** = 77

Numeric keypad divide key.

**KeypadEnter** = 82

Numeric keypad Enter key.

**KeypadMinus** = 79

Numeric keypad minus key.

**KeypadMultiply** = 78

Numeric keypad multiply key.

**KeypadPlus** = 80

Numeric keypad plus key.

**L = 94**

Letter key L.

**LastKey = 131**

The last recognized key in the enumeration.

**Left = 47**

The Left arrow key.

**M = 95**

Letter key M.

**Menu = 9**

The Menu key, typically used to open context menus.

**Minus = 120**

The Minus key.

**N = 96**

Letter key N.

**NonUsBackSlash = 130**

The Non-US backslash key.

**NumLock = 64**

The Num Lock key.

**Number0 = 109**

Number key 0.

**Number1 = 110**

Number key 1.

**Number2 = 111**

Number key 2.

**Number3 = 112**

Number key 3.

**Number4 = 113**

Number key 4.

**Number5 = 114**

Number key 5.

**Number6 = 115**

Number key 6.

**Number7 = 116**

Number key 7.

**Number8 = 117**

Number key 8.

**Number9 = 118**

Number key 9.

**O = 97**

Letter key O.

**P = 98**

Letter key P.

**PageDown = 57**

The Page Down key.

**PageUp = 56**

The Page Up key.

**Pause = 63**

The Pause key.

**Period** = 127

The Period key.

**Plus** = 121

The Plus key.

**PrintScreen** = 62

The Print Screen key.

**Q** = 99

Letter key Q.

**Quote** = 125

The Quote key.

**R** = 100

Letter key R.

**Right** = 48

The Right arrow key.

**S** = 101

Letter key S.

**ScrollLock** = 61

The Scroll Lock key.

**Semicolon** = 124

The Semicolon key.

**ShiftLeft** = 1

The left Shift key.

**ShiftRight** = 2

The right Shift key.

**Slash** = 128

The Slash key.

**Sleep** = 66

The Sleep key.

**Space** = 51

The Spacebar key.

**T** = 102

Letter key T.

**Tab** = 52

The Tab key.

**U** = 103

Letter key U.

**Unknown** = 0

Represents an unknown key.

**Up** = 45

The Up arrow key.

**V** = 104

Letter key V.

**W** = 105

Letter key W.

**WinLeft** = 7

The left Windows key.

**WinRight** = 8

The right Windows key.

X = 106

Letter key X.

Y = 107

Letter key Y.

Z = 108

Letter key Z.

# Namespace Bliss.CSharp.Interact.Mice

## Enums

[MouseButton](#)

# Enum MouseButton

Namespace: [Bliss.CSharp.Interact.Mice](#)

Assembly: Bliss.dll

```
public enum MouseButton
```

## Fields

**Left = 1**

The left mouse button.

**Middle = 2**

The middle mouse button (typically the scroll wheel button).

**Right = 3**

The right mouse button.

**X1 = 4**

The first extra (X1) mouse button.

**X2 = 5**

The second extra (X2) mouse button.

# Namespace Bliss.CSharp.Interact.Mice.Cursors

## Classes

[Sdl3Cursor](#)

## Interfaces

[ICursor](#)

## Enums

[SystemCursor](#)

# Interface ICursor

Namespace: [Bliss.CSharp.Interact.Mice.Cursors](#)

Assembly: Bliss.dll

```
public interface ICursor : IDisposable
```

## Inherited Members

[IDisposable.Dispose\(\)](#)

## Methods

### GetHandle()

Retrieves the handle of the current cursor.

```
nint GetHandle()
```

Returns

[nint](#)

A pointer to the cursor handle.

# Class Sdl3Cursor

Namespace: [Bliss.CSharp.Interact.Mice.Cursors](#)

Assembly: Bliss.dll

```
public class Sdl3Cursor : Disposable, ICursor, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← Sdl3Cursor

## Implements

[ICursor](#), [IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### Sdl3Cursor(Image, int, int)

Initializes a new instance of the [Sdl3Cursor](#) class with a custom image cursor.

```
public Sdl3Cursor(Image image, int offsetX, int offsetY)
```

## Parameters

**image** [Image](#)

The image used to create the cursor.

**offsetX** [int](#)

The X-axis offset for the cursor's hotspot.

**offsetY** [int](#)

The Y-axis offset for the cursor's hotspot.

## Sdl3Cursor(SystemCursor)

Initializes a new instance of the [Sdl3Cursor](#) class with a system cursor type.

```
public Sdl3Cursor(SystemCursor systemCursor)
```

Parameters

**systemCursor** [SystemCursor](#)

The type of system cursor to create.

## Sdl3Cursor(SDL\_Cursor\*)

Initializes a new instance of the [Sdl3Cursor](#) class with an existing SDL cursor pointer.

```
public Sdl3Cursor(SDL_Cursor* cursor)
```

Parameters

**cursor** [SDL\\_Cursor\\*](#)

A pointer to an existing SDL cursor.

## Methods

### Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

Parameters

**disposing** [bool](#)

True if called from user code; false if called from a finalizer.

## GetHandle()

Retrieves the handle of the current cursor.

```
public nint GetHandle()
```

Returns

[nint](#)

A pointer to the cursor handle.

# Enum SystemCursor

Namespace: [Bliss.CSharp.Interact.Mice.Cursors](#)

Assembly: Bliss.dll

```
public enum SystemCursor
```

## Fields

**Count** = 20

The total number of cursor types available.

**Crosshair** = 3

The crosshair cursor, typically used for precise selection.

**Default** = 0

The default system cursor.

**EResize** = 15

The resize cursor for resizing from the east side.

**EWResize** = 7

The cursor used for horizontal resizing (E-W direction).

**Move** = 9

The move cursor, typically used for dragging objects.

**NEResize** = 14

The resize cursor for resizing from the northeast corner.

**NESWResize** = 6

The resize cursor used for diagonal resizing (NE-SW direction).

**NResize** = 13

The resize cursor for resizing from the north side.

**NSResize** = 8

The cursor used for vertical resizing (N-S direction).

**NWResize** = 12

The resize cursor for resizing from the northwest corner.

**NWSEResize** = 5

The resize cursor used for diagonal resizing (NW-SE direction).

**NotAllowed** = 10

The cursor indicating an action is not allowed.

**Pointer** = 11

The pointer cursor, typically used for links and clickable items.

**Progress** = 4

The cursor indicating progress without preventing interaction.

**SEResize** = 16

The resize cursor for resizing from the southeast corner.

**SResize** = 17

The resize cursor for resizing from the south side.

**SWResize** = 18

The resize cursor for resizing from the southwest corner.

**Text** = 1

The cursor displayed when over text.

**WResize** = 19

The resize cursor for resizing from the west side.

**Wait** = 2

The cursor indicating the system is busy (wait state).



# Namespace Bliss.CSharp.Logging

## Classes

[Logger](#)

## Enums

[LogType](#)

## Delegates

[Logger.OnMessage](#)

# Enum LogType

Namespace: [Bliss.CSharp.Logging](#)

Assembly: Bliss.dll

```
public enum LogType
```

## Fields

**Debug** = 0

Debug level logging, used for detailed diagnostic messages.

**Error** = 3

Error level logging, used for error messages that indicate a failure.

**Fatal** = 4

Fatal level logging, used for critical errors that cause termination.

**Info** = 1

Info level logging, used for general informational messages.

**Warn** = 2

Warn level logging, used to indicate potential issues or warnings.

# Class Logger

Namespace: [Bliss.CSharp.Logging](#)

Assembly: Bliss.dll

```
public class Logger
```

## Inheritance

[object](#) ← Logger

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Methods

### Debug(string, int)

Logs a debug message with optional stack frame information.

```
public static void Debug(string msg, int skipFrames = 2)
```

#### Parameters

**msg** [string](#)

The debug message to be logged.

**skipFrames** [int](#)

The number of stack frames to skip (optional, default is 2).

### Error(string, int)

Logs an error message with optional stack frame information.

```
public static void Error(string msg, int skipFrames = 2)
```

## Parameters

**msg** [string](#)

The error message to be logged.

**skipFrames** [int](#)

The number of stack frames to skip (optional, default is 2).

## Fatal(Exception, int)

Logs an exception message with the color red and throws the exception.

```
public static void Fatal(Exception exception, int skipFrames = 2)
```

## Parameters

**exception** [Exception](#)

The exception to log and throw.

**skipFrames** [int](#)

The number of frames to skip when determining the source of the log message.

## Fatal(string, int)

Logs an error message and throws an exception with optional stack frame information.

```
public static void Fatal(string msg, int skipFrames = 2)
```

## Parameters

**msg** [string](#)

The fatal message to be logged.

**skipFrames** [int](#)

The number of stack frames to skip (optional, default is 2).

## Info(string, int)

Logs an informational message with optional stack frame information.

```
public static void Info(string msg, int skipFrames = 2)
```

### Parameters

msg [string](#)

The informational message to be logged.

skipFrames [int](#)

The number of stack frames to skip (optional, default is 2).

## Warn(string, int)

Logs a warning message with optional stack frame information.

```
public static void Warn(string msg, int skipFrames = 2)
```

### Parameters

msg [string](#)

The warning message to be logged.

skipFrames [int](#)

The number of stack frames to skip (optional, default is 2).

## Events

### Message

```
public static event Logger.OnMessage? Message
```

Event Type

[Logger.OnMessage](#)

# Delegate Logger.OnMessage

Namespace: [Bliss.CSharp.Logging](#)

Assembly: Bliss.dll

```
public delegate bool Logger.OnMessage(LogType type, string msg, int skipFrames,  
ConsoleColor color)
```

## Parameters

type [LogType](#)

msg [string](#)

skipFrames [int](#)

color [ConsoleColor](#)

## Returns

[bool](#)

# Namespace Bliss.CSharp.Materials

## Classes

[Material](#)

[MaterialMap](#)

[MaterialMapTypeExtensions](#)

## Enums

[MaterialMapType](#)

# Class Material

Namespace: [Bliss.CSharp.Materials](#)

Assembly: Bliss.dll

```
public class Material
```

## Inheritance

[object](#) ← Material

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Constructors

## Material(GraphicsDevice, Effect, BlendStateDescription?)

Initializes a new instance of the [Material](#) class, configuring it with the specified graphics device, shader effect, and optional blend state.

```
public Material(GraphicsDevice graphicsDevice, Effect effect, BlendStateDescription?  
blendState = null)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device to associate with this material.

**effect** [Effect](#)

The effect (shader) to apply to the material.

**blendState** [BlendStateDescription](#)?

The optional blend state to define how this material blends with others during rendering. If not specified, blending is disabled by default.

# Fields

## BlendState

Specifies the blend state for rendering, determining how colors are blended on the screen.

```
public BlendStateDescription BlendState
```

### Field Value

[BlendStateDescription](#)

## Effect

The effect (shader program) applied to this material.

```
public Effect Effect
```

### Field Value

[Effect](#)

## Parameters

A list of floating-point parameters for configuring material properties.

```
public List<float> Parameters
```

### Field Value

[List](#)<[float](#)>

## Properties

### GraphicsDevice

The graphics device associated with this material, used to manage rendering resources.

```
public GraphicsDevice GraphicsDevice { get; }
```

## Property Value

[GraphicsDevice](#)

## Methods

### AddMaterialMap(string, MaterialMap)

Adds a MaterialMap to the material's collection, associating it with a specified name.

```
public void AddMaterialMap(string name, MaterialMap map)
```

#### Parameters

**name** [string](#)

The name to associate with the MaterialMap.

**map** [MaterialMap](#)

The MaterialMap to be added to the material's collection.

### GetMapColor(string)

Retrieves the color associated with the specified material map.

```
public Color? GetMapColor(string name)
```

#### Parameters

**name** [string](#)

The name of the material map from which to retrieve the color.

Returns

[Color?](#)

The [Color](#) associated with the specified material map, or null if the map does not exist or does not have a color defined.

## GetMapTexture(string)

Retrieves the texture associated with the specified material map name.

```
public Texture2D? GetMapTexture(string name)
```

Parameters

[name string](#)

The name of the material map whose texture is to be retrieved.

Returns

[Texture2D](#)

The texture associated with the specified material map, or null if no such texture exists.

## GetMapView(string)

Gets the value associated with the specified material map name.

```
public float GetMapView(string name)
```

Parameters

[name string](#)

The name of the material map for which to retrieve the value.

Returns

[float](#)

The floating-point value associated with the specified material map name.

## GetMaterialMap(string)

Retrieves the material map associated with the specified name.

```
public MaterialMap? GetMaterialMap(string name)
```

Parameters

`name` [string](#)

The name of the material map to retrieve.

Returns

[MaterialMap](#)

The [MaterialMap](#) associated with the specified name.

## GetMaterialMapNames()

Retrieves an array of all the material map names associated with the material.

```
public string[] GetMaterialMapNames()
```

Returns

[string](#)[]

An array of strings representing the names of the material maps.

## GetMaterialMaps()

Retrieves an array of all material maps associated with the current material.

```
public MaterialMap[] GetMaterialMaps()
```

Returns

[MaterialMap\[\]](#)

An array of [MaterialMap](#) objects representing the material maps.

## GetResourceSet(Sampler, SimpleTextureLayout, string)

Retrieves the resource set associated with a specified texture in the material's map collection, based on the provided sampler, texture layout, and map name.

```
public ResourceSet? GetResourceSet(Sampler sampler, SimpleTextureLayout layout,  
string mapName)
```

Parameters

**sampler** [Sampler](#)

The sampler to use when accessing the texture.

**layout** [SimpleTextureLayout](#)

The texture layout that defines the structure of the resource set.

**mapName** [string](#)

The name of the map whose associated texture's resource set is to be retrieved.

Returns

[ResourceSet](#)

The resource set for the specified texture if found, or null if the texture does not exist in the map collection.

## SetMapColor(string, Color)

Sets the color of a specified material map.

```
public void SetMapColor(string name, Color color)
```

## Parameters

**name** [string](#)

The name of the material map whose color is to be set.

**color** [Color](#)

The color to assign to the material map.

## SetMapTexture(string, Texture2D?)

Sets the texture for the specified material map.

```
public void SetMapTexture(string name, Texture2D? texture)
```

## Parameters

**name** [string](#)

The name of the material map to set the texture for.

**texture** [Texture2D](#)

The texture to be set. If null, the material map's texture will be removed.

## SetMapView(string, float)

Sets the value of the specified material map.

```
public void SetMapView(string name, float value)
```

## Parameters

**name** [string](#)

The name of the material map to update.

**value** [float](#)

The floating-point value to set for the specified material map.

# Class MaterialMap

Namespace: [Bliss.CSharp.Materials](#)

Assembly: Bliss.dll

```
public class MaterialMap
```

## Inheritance

[object](#) ← MaterialMap

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Constructors

## MaterialMap(Texture2D?, Color?, float)

Initializes a new instance of the [MaterialMap](#) class with the specified texture, color, and value.

```
public MaterialMap(Texture2D? texture = null, Color? color = null, float value = 0)
```

## Parameters

**texture** [Texture2D](#)

The texture associated with the material map. Can be [null](#).

**color** [Color](#)?

The color associated with the material map. Can be [null](#).

**value** [float](#)

The scalar value associated with the material map, defaulting to [0.0F](#).

# Fields

# Color

Represents color information with properties for red, green, blue, and alpha components. This structure is used for defining the color attributes of various graphical elements such as materials and textures.

```
public Color? Color
```

## Field Value

[Color?](#)

# Texture

Represents a 2D texture used in a material map. This texture can include properties such as height, width, pixel format, and mip levels. It is used for rendering purposes in a graphics device.

```
public Texture2D? Texture
```

## Field Value

[Texture2D](#)

# Value

Represents a float value associated with a material map. This value can be used for various purposes such as setting shader parameters or controlling material properties in a rendering engine.

```
public float Value
```

## Field Value

[float](#)

# Enum MaterialMapType

Namespace: [Bliss.CSharp.Materials](#)

Assembly: Bliss.dll

```
public enum MaterialMapType
```

## Extension Methods

[MaterialMapTypeExtensions.GetName\(MaterialMapType\)](#)

## Fields

[**EnumMember**(Value = "fAlbedo")] Albedo = 0

Albedo map, which defines the base color of the material.

[**EnumMember**(Value = "fEmissive")] Emission = 5

Emission map, which defines areas of the material that emit light.

[**EnumMember**(Value = "fHeight")] Height = 6

Height map, which provides height data for the material to simulate depth effects.

[**EnumMember**(Value = "fMetallic")] Metallic = 1

Metallic map, which defines the metallic properties of the material.

[**EnumMember**(Value = "fNormal")] Normal = 2

Normal map, which adds detail to the surface of the material by simulating bumps and grooves.

[**EnumMember**(Value = "fOcclusion")] Occlusion = 4

Occlusion map, which adds shadows in crevices to enhance the appearance of depth.

[**EnumMember**(Value = "fRoughness")] Roughness = 3

Roughness map, which defines the roughness level of the material's surface.

# Class MaterialMapTypeExtensions

Namespace: [Bliss.CSharp.Materials](#)

Assembly: Bliss.dll

```
public static class MaterialMapTypeExtensions
```

## Inheritance

[object](#) ← MaterialMapTypeExtensions

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Methods

## GetName(MaterialMapType)

Retrieves the string representation of the specified [MaterialMapType](#) enum value, using the value from the [EnumMemberAttribute](#) if available. If the attribute is not present, it returns the enum value's name as a string.

```
public static string GetName(this MaterialMapType type)
```

### Parameters

**type** [MaterialMapType](#)

The [MaterialMapType](#) enum value for which to retrieve the name.

### Returns

[string](#)

The string representation of the [MaterialMapType](#). If the [EnumMemberAttribute](#) is defined, its value is returned; otherwise, the enum name is returned.

# Namespace Bliss.CSharp.Mathematics

## Classes

[BlissMath](#)

# Class BlissMath

Namespace: [Bliss.CSharp.Mathematics](#)

Assembly: Bliss.dll

```
public static class BlissMath
```

## Inheritance

[object](#) ← BlissMath

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Methods

## Vector3Angle(Vector3, Vector3)

Calculates the angle in radians between two vectors.

```
public static float Vector3Angle(Vector3 v1, Vector3 v2)
```

### Parameters

v1 [Vector3](#)

The first vector.

v2 [Vector3](#)

The second vector.

### Returns

[float](#)

The angle in radians between the two vectors.

## Vector3RotateByAxisAngle(Vector3, Vector3, float)

Rotates a vector around a specified axis by a given angle.

```
public static Vector3 Vector3RotateByAxisAngle(Vector3 v, Vector3 axis, float angle)
```

### Parameters

v [Vector3](#)

The vector to be rotated.

axis [Vector3](#)

The axis around which to rotate the vector.

angle [float](#)

The angle in radians by which to rotate the vector.

### Returns

[Vector3](#)

The rotated vector.

# Namespace Bliss.CSharp.Textures

## Classes

[MipmapHelper](#)

[RenderTexture2D](#)

[Texture2D](#)

# Class MipmapHelper

Namespace: [Bliss.CSharp.Textures](#)

Assembly: Bliss.dll

```
public static class MipmapHelper
```

## Inheritance

[object](#) ← MipmapHelper

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Methods

## GenerateMipmaps(Image)

Generates mipmaps for a given source image.

```
public static Image[] GenerateMipmaps(Image baseImage)
```

### Parameters

**baseImage** [Image](#)

The base image from which the mipmaps are generated.

### Returns

[Image](#)[]

A list of images representing the mipmap levels, including the original image as the first level.

# Class RenderTexture2D

Namespace: [Bliss.CSharp.Textures](#)

Assembly: Bliss.dll

```
public class RenderTexture2D : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← RenderTexture2D

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### RenderTexture2D(GraphicsDevice, uint, uint, TextureSampleCount)

Initializes a new instance of the [RenderTexture2D](#) class, creating a render target texture with specified dimensions and sample count.

```
public RenderTexture2D(GraphicsDevice graphicsDevice, uint width, uint height,  
TextureSampleCount sampleCount = TextureSampleCount.Count1)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The [GraphicsDevice](#) used to create and manage the texture.

**width** [uint](#)

The width of the render texture in pixels.

## `height` [uint](#)

The height of the render texture in pixels.

## `sampleCount` [TextureSampleCount](#)

The number of samples for multisampling. Defaults to [Count1](#).

# Properties

## ColorTexture

Gets the color texture used for rendering operations within the render texture.

```
public Texture ColorTexture { get; }
```

### Property Value

[Texture](#)

## DepthTexture

Gets the depth texture associated with this render texture, used for depth and stencil operations.

```
public Texture DepthTexture { get; }
```

### Property Value

[Texture](#)

## DestinationTexture

Gets the destination texture used for sampling operations in this render texture.

```
public Texture DestinationTexture { get; }
```

### Property Value

## Framebuffer

Gets the framebuffer used for rendering to the textures associated with this render texture.

```
public Framebuffer Framebuffer { get; }
```

Property Value

[Framebuffer](#)

## GraphicsDevice

Gets the graphics device associated with this render texture.

```
public GraphicsDevice GraphicsDevice { get; }
```

Property Value

[GraphicsDevice](#)

## Height

Gets the height of the render texture.

```
public uint Height { get; }
```

Property Value

[uint](#)

## SampleCount

Gets or sets the sample count for the render texture.

```
public TextureSampleCount SampleCount { get; set; }
```

## Property Value

[TextureSampleCount](#)

## Width

Gets the width of the render texture.

```
public uint Width { get; }
```

## Property Value

[uint](#)

## Methods

### CreateFrameBuffer()

Creates a framebuffer with depth and color textures based on the specified width, height, and sample count.

```
public void CreateFrameBuffer()
```

### Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

## Parameters

[disposing](#) [bool](#)

True if called from user code; false if called from a finalizer.

## GetResourceSet(Sampler, ResourceLayout)

Retrieves a [ResourceSet](#) for the specified [Sampler](#) and [ResourceLayout](#). If the resource set is not already cached, a new one is created, cached, and then returned.

```
public ResourceSet GetResourceSet(Sampler sampler, ResourceLayout layout)
```

### Parameters

**sampler** [Sampler](#)

The sampler to be used in the resource set.

**layout** [ResourceLayout](#)

The resource layout defining how resources are bound to the pipeline.

### Returns

[ResourceSet](#)

A [ResourceSet](#) that contains the specified sampler and layout.

## Resize(uint, uint)

Resizes the render textures and framebuffer to the new specified width and height.

```
public void Resize(uint width, uint height)
```

### Parameters

**width** [uint](#)

The new width of the render texture.

**height** [uint](#)

The new height of the render texture.

# Class Texture2D

Namespace: [Bliss.CSharp.Textures](#)

Assembly: Bliss.dll

```
public class Texture2D : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← Texture2D

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### Texture2D(GraphicsDevice, Image, bool, bool)

Initializes a new instance of the [Texture2D](#) class using an [Image](#) object. Creates a device texture, applies the specified sampler, and stores mipmap and color format information.

```
public Texture2D(GraphicsDevice graphicsDevice, Image image, bool mipmap = true, bool srgb = false)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device used for rendering the texture.

**image** [Image](#)

The image object used to create the texture.

**mipmap** [bool](#)

Indicates whether to generate mipmaps for the texture.

#### srgb [bool](#)

Indicates whether to use sRGB color space for the texture.

## Texture2D(GraphicsDevice, Stream, bool, bool)

Initializes a new instance of the [Texture2D](#) class using an image stream. Loads the texture from the specified stream and applies the given sampler, mipmap, and sRGB settings.

```
public Texture2D(GraphicsDevice graphicsDevice, Stream stream, bool mipmap = true, bool srgb = false)
```

### Parameters

#### graphicsDevice [GraphicsDevice](#)

The graphics device used for rendering the texture.

#### stream [Stream](#)

The image stream to load as a texture.

#### mipmap [bool](#)

Indicates whether to generate mipmaps for the texture.

#### srgb [bool](#)

Indicates whether to use sRGB color space for the texture.

## Texture2D(GraphicsDevice, string, bool, bool)

Initializes a new instance of the [Texture2D](#) class using an image file path. Loads the texture from the specified path and applies the given sampler, mipmap, and sRGB settings.

```
public Texture2D(GraphicsDevice graphicsDevice, string path, bool mipmap = true, bool srgb = false)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device used for rendering the texture.

**path** [string](#)

The file path of the image to load as a texture.

**mipmap** [bool](#)

Indicates whether to generate mipmaps for the texture.

**srgb** [bool](#)

Indicates whether to use sRGB color space for the texture.

## Properties

### DeviceTexture

Gets the device texture created from the images.

```
public Texture DeviceTexture { get; }
```

### Property Value

[Texture](#)

### Format

Gets the pixel format of the texture.

```
public PixelFormat Format { get; }
```

### Property Value

[PixelFormat](#)

## GraphicsDevice

Gets the graphics device associated with this texture.

```
public GraphicsDevice GraphicsDevice { get; }
```

Property Value

[GraphicsDevice](#)

## Height

Gets the height of the texture in pixels.

```
public uint Height { get; }
```

Property Value

[uint](#)

## Images

Gets the array of images representing the mip levels of the texture.

```
public Image[] Images { get; }
```

Property Value

[Image\[\]](#)

## MipLevels

Gets the number of mip levels in the texture.

```
public uint MipLevels { get; }
```

Property Value

[uint](#)

## PixelSizeInBytes

Gets the size of a pixel in bytes.

```
public uint PixelSizeInBytes { get; }
```

Property Value

[uint](#)

## Width

Gets the width of the texture in pixels.

```
public uint Width { get; }
```

Property Value

[uint](#)

## Methods

### Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

Parameters

`disposing` [bool](#)

True if called from user code; false if called from a finalizer.

## GetDataFromBytes()

Retrieves the texture data as a byte array.

```
public byte[] GetDataFromBytes()
```

Returns

[byte](#)

A byte array containing the texture data.

## GetDataFromImage()

Retrieves the image data from the first image in the texture array.

```
public Image GetDataFromImage()
```

Returns

[Image](#)

An [Image](#) representing the image data.

## GetResourceSet(Sampler, SimpleTextureLayout)

Gets a resource set associated with the specified sampler and resource layout. If the resource set is already cached, it returns the cached resource set; otherwise, it creates and caches a new one.

```
public ResourceSet GetResourceSet(Sampler sampler, SimpleTextureLayout layout)
```

Parameters

sampler [Sampler](#)

The sampler used for the resource set.

layout [SimpleTextureLayout](#)

The resource layout used for the resource set.

Returns

[ResourceSet](#)

The resource set associated with the specified sampler and resource layout.

## SetData(Image)

Loads the provided image data into the texture. If the image dimensions do not match the texture dimensions, an exception is thrown. The image data is then split into mip levels if required and updated in the device texture.

```
public void SetData(Image data)
```

Parameters

**data** [Image](#)

The image data to be loaded into the texture.

## SetData(byte[], Rectangle?)

Sets the texture data from a byte array, optionally specifying a rectangular area within the texture to update.

```
public void SetData(byte[] data, Rectangle? area = null)
```

Parameters

**data** [byte](#)[]

The byte array containing the texture data.

**area** [Rectangle](#)?

The rectangular area within the texture to update. If null, it updates the entire texture.



# Namespace Bliss.CSharp.Textures.Cubemaps

## Classes

[Cubemap](#)

[CubemapHelper](#)

## Enums

[CubemapLayer](#)

[CubemapLayout](#)

# Class Cubemap

Namespace: [Bliss.CSharp.Textures.Cubemaps](#)

Assembly: Bliss.dll

```
public class Cubemap : Disposable, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← Cubemap

## Implements

[IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

# Constructors

## Cubemap(GraphicsDevice, Image, CubemapLayout, bool, bool)

Initializes a new instance of the [Cubemap](#) class by splitting a single image into cubemap faces.

```
public Cubemap(GraphicsDevice graphicsDevice, Image image, CubemapLayout layout =  
    CubemapLayout.AutoDetect, bool mipmap = true, bool srgb = false)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device used for creating resources.

**image** [Image](#)

The source image containing all cubemap faces in the specified layout.

**layout** [CubemapLayout](#)

The layout format used to interpret the cubemap faces in the source image.

#### mipmap [bool](#)

Specifies whether to generate mipmaps for each cubemap face.

#### srgb [bool](#)

Specifies whether the images should be loaded as sRGB format.

## Cubemap(GraphicsDevice, Stream, CubemapLayout, bool, bool)

Initializes a new instance of the [Cubemap](#) class by loading cubemap faces from a stream.

```
public Cubemap(GraphicsDevice graphicsDevice, Stream stream, CubemapLayout layout =  
    CubemapLayout.AutoDetect, bool mipmap = true, bool srgb = false)
```

### Parameters

#### graphicsDevice [GraphicsDevice](#)

The graphics device used for creating resources.

#### stream [Stream](#)

The stream containing the image data for the cubemap faces.

#### layout [CubemapLayout](#)

The layout of the cubemap faces within the image.

#### mipmap [bool](#)

Specifies whether to generate mipmaps for the cubemap.

#### srgb [bool](#)

Specifies whether to load the image as sRGB.

## Cubemap(GraphicsDevice, string, CubemapLayout, bool, bool)

Initializes a new instance of the [Cubemap](#) class by loading cubemap faces from a file path.

```
public Cubemap(GraphicsDevice graphicsDevice, string path, CubemapLayout layout = CubemapLayout.AutoDetect, bool mipmap = true, bool srgb = false)
```

## Parameters

**graphicsDevice** [GraphicsDevice](#)

The graphics device used for creating resources.

**path** [string](#)

The file path to the image containing cubemap faces.

**layout** [CubemapLayout](#)

The layout of the cubemap faces within the image.

**mipmap** [bool](#)

Specifies whether to generate mipmaps for the cubemap.

**srgb** [bool](#)

Specifies whether to load the image as sRGB.

## Properties

### DeviceTexture

Gets the device texture.

```
public Texture DeviceTexture { get; }
```

### Property Value

[Texture](#)

### Format

Gets the pixel format of the cubemap.

```
public PixelFormat Format { get; }
```

Property Value

[PixelFormat](#)

## GraphicsDevice

Gets the graphics device associated with the cubemap instance.

```
public GraphicsDevice GraphicsDevice { get; }
```

Property Value

[GraphicsDevice](#)

## Height

Gets the height of the cubemap.

```
public uint Height { get; }
```

Property Value

[uint](#)

## Images

Gets the mipmap levels of images for each face of the cubemap.

```
public Image[][] Images { get; }
```

Property Value

[Image](#)[][]

## MipLevels

Gets the number of mip levels of the cubemap.

```
public uint MipLevels { get; }
```

Property Value

[uint](#)

## PixelSizeInBytes

Gets the size of a pixel in bytes.

```
public uint PixelSizeInBytes { get; }
```

Property Value

[uint](#)

## TextureView

Gets the texture view associated with the cubemap, allowing shaders to access the cubemap texture data.

```
public TextureView TextureView { get; }
```

Property Value

[TextureView](#)

## Width

Gets the width of the cubemap.

```
public uint Width { get; }
```

Property Value

[uint](#)

## Methods

### Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

Parameters

[disposing](#) [bool](#)

True if called from user code; false if called from a finalizer.

### GetDataFromBytes(CubemapLayer)

Retrieves the raw byte data for a specific array layer of the cubemap.

```
public byte[] GetDataFromBytes(CubemapLayer cubemapLayer)
```

Parameters

[cubemapLayer](#) [CubemapLayer](#)

The array layer of the cubemap to retrieve data from.

Returns

[byte](#)[]

An array of bytes containing the pixel data for the specified array layer.

### GetDataFromImage(CubemapLayer)

Retrieves the image data associated with the specified array layer of the cubemap.

```
public Image GetDataFromImage(CubemapLayer cubemapLayer)
```

## Parameters

cubemapLayer [CubemapLayer](#)

The specific array layer whose image data is to be retrieved.

## Returns

[Image](#)

An [Image](#) object containing the image data for the specified array layer.

## GetResourceSet(Sampler, ResourceLayout)

Retrieves a resource set from the cache or creates a new one using a specified sampler and texture layout.

```
public ResourceSet GetResourceSet(Sampler sampler, ResourceLayout layout)
```

## Parameters

sampler [Sampler](#)

The sampler object to be used for the resource set.

layout [ResourceLayout](#)

The texture layout to be used for the resource set.

## Returns

[ResourceSet](#)

A [ResourceSet](#) object associated with the provided sampler and layout.

## SetData(Image, CubemapLayer)

Updates the data of a specified layer in the cubemap with the provided image data.

```
public void SetData(Image data, CubemapLayer cubemapLayer)
```

### Parameters

**data** [Image](#)

The image data to set, which must match the dimensions of the texture.

**cubemapLayer** [CubemapLayer](#)

The layer of the cubemap to update.

### Exceptions

[ArgumentException](#)

Throws if the dimensions of the provided image data do not match the dimensions of the cubemap.

## SetData(byte[], CubemapLayer, Rectangle?)

Updates the cubemap's data for a specified array layer and a rectangular area.

```
public void SetData(byte[] data, CubemapLayer cubemapLayer, Rectangle? area = null)
```

### Parameters

**data** [byte](#)[]

The raw byte array containing the pixel data to be set.

**cubemapLayer** [CubemapLayer](#)

The specific cubemap array layer to update.

**area** [Rectangle](#)?

An optional rectangle specifying the area to update. If null, the whole layer will be updated.

# Class CubemapHelper

Namespace: [Bliss.CSharp.Textures.Cubemaps](#)

Assembly: Bliss.dll

```
public static class CubemapHelper
```

## Inheritance

[object](#) ← CubemapHelper

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Methods

### GenCubemapImages(Image, CubemapLayout)

Generates an array of images, each representing one face of a cubemap, based on the input image and specified cubemap layout.

```
public static Image[] GenCubemapImages(Image image, CubemapLayout layout)
```

#### Parameters

**image** [Image](#)

The input image to generate cubemap face images from.

**layout** [CubemapLayout](#)

The layout of the cubemap in the source image. Determines how the image should be split into faces.

#### Returns

[Image](#)[]

An array of six images, each representing one face of the cubemap. The order of the images corresponds to the cubemap faces.

## Exceptions

### [ArgumentException](#)

Thrown when the dimensions of the input image are incompatible with the specified cubemap layout.

### [ArgumentOutOfRangeException](#)

Thrown when the specified cubemap layout is not recognized or supported.

# Enum CubemapLayer

Namespace: [Bliss.CSharp.Textures.Cubemaps](#)

Assembly: Bliss.dll

```
public enum CubemapLayer
```

## Fields

**NegativeX = 1**

The negative X face of the cubemap (left).

**NegativeY = 3**

The negative Y face of the cubemap (bottom).

**NegativeZ = 5**

The negative Z face of the cubemap (back).

**PositiveX = 0**

The positive X face of the cubemap (right).

**PositiveY = 2**

The positive Y face of the cubemap (top).

**PositiveZ = 4**

The positive Z face of the cubemap (front).

# Enum CubemapLayout

Namespace: [Bliss.CSharp.Textures.Cubemaps](#)

Assembly: Bliss.dll

```
public enum CubemapLayout
```

## Fields

**AutoDetect = 0**

Automatically detects the cubemap layout based on the texture dimensions and aspect ratio.

**CrossFourByThree = 4**

Specifies a 4x3 cross layout for cubemaps, with four columns and three rows of faces.

**CrossThreeByFour = 3**

Specifies a 3x4 cross layout for cubemaps, with three columns and four rows of faces.

**LineHorizontal = 2**

Specifies a horizontal line layout where cubemap faces are aligned horizontally in a single row.

**LineVertical = 1**

Specifies a vertical line layout where cubemap faces are stacked vertically in a single column.

**Panorama = 5**

Specifies a panoramic layout, used for spherical or equirectangular textures that can be converted to cubemaps.

# Namespace Bliss.CSharp.Transformations

## Structs

[Point](#)

[Rectangle](#)

[RectangleF](#)

[Transform](#)

# Struct Point

Namespace: [Bliss.CSharp.Transformations](#)

Assembly: Bliss.dll

```
public struct Point : IEquatable<Point>
```

Implements

[IEquatable](#)<[Point](#)>

Inherited Members

[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### Point(int, int)

Initializes a new instance of the Point class with the specified X and Y coordinates.

```
public Point(int x, int y)
```

Parameters

x [int](#)

The X coordinate of the point.

y [int](#)

The Y coordinate of the point.

## Fields

### X

The X-coordinate of the Point structure. Represents the horizontal component in a 2D space.

```
public int X
```

Field Value

[int](#)

Y

The Y-coordinate of the Point structure. Represents the vertical component in a 2D space.

```
public int Y
```

Field Value

[int](#)

## Methods

### Equals(Point)

Indicates whether this instance and a specified Point object represent the same point.

```
public bool Equals(Point other)
```

Parameters

**other** [Point](#)

The Point to compare with the current instance.

Returns

[bool](#)

true if the specified Point is equal to the current Point; otherwise, false.

## Equals(object?)

Determines whether this instance and a specified object represent the same point.

```
public override bool Equals(object? obj)
```

Parameters

**obj** [object](#)

The object to compare with the current instance.

Returns

[bool](#)

true if the specified object is a Point and is equal to the current Point; otherwise, false.

## GetHashCode()

Returns the hash code for this Point instance.

```
public override int GetHashCode()
```

Returns

[int](#)

A hash code for the current Point, which is a unique identifier of the instance.

## ToString()

Returns a string that represents the current Point instance.

```
public override string ToString()
```

Returns

## [string](#)

A string that contains the X and Y coordinates of the Point instance.

# Operators

## operator ==(Point, Point)

Determines whether two specified Point instances have the same coordinates.

```
public static bool operator ==(Point left, Point right)
```

Parameters

### **left** [Point](#)

The first Point to compare.

### **right** [Point](#)

The second Point to compare.

Returns

## [bool](#)

true if both Point instances have the same X and Y coordinates; otherwise, false.

## operator !=(Point, Point)

Determines whether two specified Point instances have different coordinates.

```
public static bool operator !=(Point left, Point right)
```

Parameters

### **left** [Point](#)

The first Point to compare.

## **right** Point

The second Point to compare.

Returns

bool ↗

true if the Point instances do not have the same X and Y coordinates; otherwise, false.

# Struct Rectangle

Namespace: [Bliss.CSharp.Transformations](#)

Assembly: Bliss.dll

```
public struct Rectangle : IEquatable<Rectangle>
```

Implements

[IEquatable](#)<[Rectangle](#)>

Inherited Members

[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### Rectangle(int, int, int, int)

Initializes a new instance of the [Rectangle](#) struct with the specified position and size.

```
public Rectangle(int x, int y, int width, int height)
```

Parameters

x [int](#)

The X-coordinate of the rectangle's top-left corner.

y [int](#)

The Y-coordinate of the rectangle's top-left corner.

width [int](#)

The width of the rectangle.

height [int](#)

The height of the rectangle.

# Fields

## Height

The height of the rectangle.

```
public int Height
```

### Field Value

[int ↗](#)

## Width

The width of the rectangle.

```
public int Width
```

### Field Value

[int ↗](#)

## X

The X-coordinate of the rectangle's top-left corner.

```
public int X
```

### Field Value

[int ↗](#)

## Y

The Y-coordinate of the rectangle's top-left corner.

```
public int Y
```

Field Value

[int ↗](#)

## Properties

### Position

Gets or sets the position (X, Y) of the rectangle's top-left corner.

```
public Vector2 Position { get; set; }
```

Property Value

[Vector2 ↗](#)

### Size

Gets or sets the size (Width, Height) of the rectangle.

```
public Vector2 Size { get; set; }
```

Property Value

[Vector2 ↗](#)

## Methods

### Contains(Vector2)

Determines whether the specified point is contained within the rectangle.

```
public bool Contains(Vector2 p)
```

## Parameters

p [Vector2](#)

The point to check.

## Returns

[bool](#)

True if the point is inside the rectangle; otherwise, false.

## Contains(float, float)

Determines whether the specified coordinates are contained within the rectangle.

```
public bool Contains(float x, float y)
```

## Parameters

x [float](#)

The X-coordinate of the point to check.

y [float](#)

The Y-coordinate of the point to check.

## Returns

[bool](#)

True if the point is inside the rectangle; otherwise, false.

## Equals(Rectangle)

Determines whether the current rectangle is equal to another [Rectangle](#).

```
public bool Equals(Rectangle other)
```

## Parameters

### other [Rectangle](#)

The rectangle to compare with the current rectangle.

## Returns

### [bool](#) ↗

True if the rectangles are equal; otherwise, false.

## Equals(object?)

Determines whether the specified object is equal to the current rectangle.

```
public override bool Equals(object? obj)
```

## Parameters

### obj [object](#) ↗

The object to compare with the current rectangle.

## Returns

### [bool](#) ↗

True if the object is a [Rectangle](#) and is equal to the current rectangle; otherwise, false.

## GetHashCode()

Returns a hash code for the current rectangle.

```
public override int GetHashCode()
```

## Returns

### [int](#) ↗

A hash code for the current rectangle.

## ToString()

Returns a string that represents the current rectangle.

```
public override string ToString()
```

Returns

[string](#)

A string that represents the rectangle's position and size.

## Operators

### operator ==(Rectangle, Rectangle)

Determines whether two [Rectangle](#) instances are equal.

```
public static bool operator ==(Rectangle left, Rectangle right)
```

Parameters

**left** [Rectangle](#)

The first [Rectangle](#) to compare.

**right** [Rectangle](#)

The second [Rectangle](#) to compare.

Returns

[bool](#)

True if both rectangles are equal; otherwise, false.

## operator !=(Rectangle, Rectangle)

Determines whether two [Rectangle](#) instances are not equal.

```
public static bool operator !=(Rectangle left, Rectangle right)
```

### Parameters

**left** [Rectangle](#)

The first [Rectangle](#) to compare.

**right** [Rectangle](#)

The second [Rectangle](#) to compare.

### Returns

[bool](#) ↗

True if the rectangles are not equal; otherwise, false.

# Struct RectangleF

Namespace: [Bliss.CSharp.Transformations](#)

Assembly: Bliss.dll

```
public struct RectangleF : IEquatable<RectangleF>
```

Implements

[IEquatable](#)<[RectangleF](#)>

Inherited Members

[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### RectangleF(float, float, float, float)

Initializes a new instance of the [RectangleF](#) struct with the specified position and size.

```
public RectangleF(float x, float y, float width, float height)
```

Parameters

**x** [float](#)

The X-coordinate of the rectangle's top-left corner.

**y** [float](#)

The Y-coordinate of the rectangle's top-left corner.

**width** [float](#)

The width of the rectangle.

**height** [float](#)

The height of the rectangle.

# Fields

## Height

The height of the rectangle.

```
public float Height
```

### Field Value

[float](#)

## Width

The width of the rectangle.

```
public float Width
```

### Field Value

[float](#)

## X

The X-coordinate of the rectangle's top-left corner.

```
public float X
```

### Field Value

[float](#)

## Y

The Y-coordinate of the rectangle's top-left corner.

```
public float Y
```

Field Value

[float](#)

## Properties

### Position

Gets or sets the position (X, Y) of the rectangle's top-left corner.

```
public Vector2 Position { get; set; }
```

Property Value

[Vector2](#)

### Size

Gets or sets the size (Width, Height) of the rectangle.

```
public Vector2 Size { get; set; }
```

Property Value

[Vector2](#)

## Methods

### Contains(Vector2)

Determines whether the specified point is contained within the rectangle.

```
public bool Contains(Vector2 p)
```

## Parameters

p [Vector2](#)

The point to check.

## Returns

[bool](#)

True if the point is inside the rectangle; otherwise, false.

## Contains(float, float)

Determines whether the specified coordinates are contained within the rectangle.

```
public bool Contains(float x, float y)
```

## Parameters

x [float](#)

The X-coordinate of the point to check.

y [float](#)

The Y-coordinate of the point to check.

## Returns

[bool](#)

True if the point is inside the rectangle; otherwise, false.

## Equals(RectangleF)

Determines whether the current rectangle is equal to another [RectangleF](#).

```
public bool Equals(RectangleF other)
```

## Parameters

**other** [RectangleF](#)

The rectangle to compare with the current rectangle.

## Returns

[bool](#) ↗

True if the rectangles are equal; otherwise, false.

## Equals(object?)

Determines whether the specified object is equal to the current rectangle.

```
public override bool Equals(object? obj)
```

## Parameters

**obj** [object](#) ↗

The object to compare with the current rectangle.

## Returns

[bool](#) ↗

True if the object is a [RectangleF](#) and is equal to the current rectangle; otherwise, false.

## GetHashCode()

Returns a hash code for the current rectangle.

```
public override int GetHashCode()
```

## Returns

[int](#) ↗

A hash code for the current rectangle.

## ToString()

Returns a string that represents the current rectangle.

```
public override string ToString()
```

Returns

[string](#)

A string that represents the rectangle's position and size.

## Operators

### operator ==(RectangleF, RectangleF)

Determines whether two [RectangleF](#) instances are equal.

```
public static bool operator ==(RectangleF left, RectangleF right)
```

Parameters

**left** [RectangleF](#)

The first [RectangleF](#) to compare.

**right** [RectangleF](#)

The second [RectangleF](#) to compare.

Returns

[bool](#)

True if both rectangles are equal; otherwise, false.

## operator !=(RectangleF, RectangleF)

Determines whether two [RectangleF](#) instances are not equal.

```
public static bool operator !=(RectangleF left, RectangleF right)
```

### Parameters

**left** [RectangleF](#)

The first [RectangleF](#) to compare.

**right** [RectangleF](#)

The second [RectangleF](#) to compare.

### Returns

[bool](#) ↗

True if the rectangles are not equal; otherwise, false.

# Struct Transform

Namespace: [Bliss.CSharp.Transformations](#)

Assembly: Bliss.dll

```
public struct Transform : IEquatable<Transform>
```

Implements

[IEquatable](#)<[Transform](#)>

Inherited Members

[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### Transform()

Initializes a new instance of the [Transform](#) class with default values.

```
public Transform()
```

## Fields

### Rotation

Represents the rotation of an object in 3D space.

```
public Quaternion Rotation
```

### Field Value

[Quaternion](#)

### Scale

Represents the scale of an object in 3D space.

```
public Vector3 Scale
```

Field Value

[Vector3](#)

## Translation

Represents the position of an object in 3D space.

```
public Vector3 Translation
```

Field Value

[Vector3](#)

## Methods

### Equals(Transform)

Determines whether the current instance is equal to another instance of the [Transform](#) struct.

```
public bool Equals(Transform other)
```

Parameters

**other** [Transform](#)

The [Transform](#) to compare with the current instance.

Returns

[bool](#)

**true** if the current instance is equal to the specified [Transform](#); otherwise, **false**.

## Equals(object?)

Determines whether the specified object is equal to the current [Transform](#) instance.

```
public override bool Equals(object? obj)
```

Parameters

**obj** [object](#)

The object to compare with the current [Transform](#).

Returns

[bool](#)

**true** if the specified object is equal to the current [Transform](#); otherwise, **false**.

## GetHashCode()

Returns the hash code for the current instance of the [Transform](#) struct.

```
public override int GetHashCode()
```

Returns

[int](#)

An integer representing the hash code of the current [Transform](#) instance.

## GetTransform()

Returns the transformation matrix for the current Transform object.

```
public Matrix4x4 GetTransform()
```

Returns

## [Matrix4x4](#)

The transformation matrix.

### **ToString()**

Returns a string representation of the current Transform object.

```
public override string ToString()
```

Returns

[string](#)

A string that represents the values of Translation, Rotation, and Scale.

## **Operators**

### **operator ==(Transform, Transform)**

Determines whether two instances of the [Transform](#) struct are equal.

```
public static bool operator ==(Transform left, Transform right)
```

Parameters

**left** [Transform](#)

The first instance of [Transform](#) to compare.

**right** [Transform](#)

The second instance of [Transform](#) to compare.

Returns

[bool](#)

**true** if the two instances are equal; otherwise, **false**.

# operator !=(Transform, Transform)

Determines whether two instances of the [Transform](#) struct are not equal.

```
public static bool operator !=(Transform left, Transform right)
```

## Parameters

**left** [Transform](#)

The first instance of [Transform](#) to compare.

**right** [Transform](#)

The second instance of [Transform](#) to compare.

## Returns

[bool](#) ↗

**true** if the two instances are not equal; otherwise, **false**.

# Namespace Bliss.CSharp.Windowing

## Classes

[Sdl3Window](#)

[Window](#)

## Interfaces

[IWindow](#)

## Enums

[WindowState](#)

[WindowType](#)

# Interface IWindow

Namespace: [Bliss.CSharp.Windowing](#)

Assembly: Bliss.dll

```
public interface IWindow : IDisposable
```

## Inherited Members

[IDisposable.Dispose\(\)](#) ↗

## Properties

### Exists

Indicates whether the window currently exists. This property can be used to check if the window has not been closed or destroyed.

```
bool Exists { get; }
```

#### Property Value

[bool](#) ↗

### Handle

Gets the native handle of the window. This handle can be used to interact with low-level windowing operations directly through platform-specific APIs or libraries.

```
nint Handle { get; }
```

#### Property Value

[nint](#) ↗

## Id

Gets the unique identifier for the window. This identifier can be used to reference the window in various windowing and event handling operations.

```
uint Id { get; }
```

### Property Value

[uint](#)

## IsFocused

Indicates whether the window is currently focused. A window is considered focused when it has received input focus and is the active window receiving user input. This property can be used to check if the window is the foreground window.

```
bool IsFocused { get; }
```

### Property Value

[bool](#)

## SwapchainSource

Gets the source of the swapchain tied to the window. This source provides the necessary information to create and manage swapchains for rendering graphics content within the window.

```
SwapchainSource SwapchainSource { get; }
```

### Property Value

[SwapchainSource](#)

## Methods

## ClearState()

Clears the current state of the window, resetting it to its default state.

```
void ClearState()
```

## ClientToScreen(Point)

Converts the specified client-area point to screen coordinates.

```
Point ClientToScreen(Point point)
```

Parameters

point [Point](#)

The client-area point to be converted.

Returns

[Point](#)

The converted point in screen coordinates.

## GetHeight()

Retrieves the current height of the window.

```
int GetHeight()
```

Returns

[int](#)

The current window height as an integer.

## GetOrCreateOpenGLPlatformInfo(GraphicsDeviceOptions, GraphicsBackend)

Retrieves or creates the OpenGL platform information required for the specified graphics device options and backend.

```
OpenGLPlatformInfo GetOrCreateOpenGLPlatformInfo(GraphicsDeviceOptions options,  
GraphicsBackend backend)
```

### Parameters

**options** [GraphicsDeviceOptions](#)

The graphics device options to use.

**backend** [GraphicsBackend](#)

The graphics backend to use.

### Returns

[OpenGLPlatformInfo](#)

The OpenGL platform information.

## GetPosition()

Retrieves the current position of the window.

```
(int, int) GetPosition()
```

### Returns

[\(int, int\)](#)

A tuple containing the x and y coordinates of the window.

## GetSize()

Retrieves the current size of the window.

```
(int, int) GetSize()
```

Returns

[\(int\)](#), [\(int\)](#)

A tuple containing the width and height of the window.

## GetState()

Retrieves the current state of the window.

```
WindowState GetState()
```

Returns

[WindowState](#)

The current state of the window as a `WindowState` enum value.

## GetTitle()

Retrieves the current title of the window.

```
string GetTitle()
```

Returns

[string](#)

The current window title as a string.

## GetWidth()

Retrieves the current width of the window.

```
int GetWidth()
```

Returns

[int↗](#)

The current window width as an integer.

## GetX()

Retrieves the current X coordinate of the window.

```
int GetX()
```

Returns

[int↗](#)

The current X coordinate of the window as an integer.

## GetY()

Retrieves the current y-coordinate of the window's position.

```
int GetY()
```

Returns

[int↗](#)

The current y-coordinate of the window as an integer.

## HasState(WindowState)

Checks if the current state of the window matches the specified state.

```
bool HasState(WindowState state)
```

## Parameters

state [WindowState](#)

The state to be checked against the current window state.

## Returns

[bool](#)

True if the current state of the window matches the specified state, otherwise false.

## PumpEvents()

Processes all pending window events.

```
void PumpEvents()
```

## ScreenToClient(Point)

Converts the specified screen coordinates to client-area coordinates.

```
Point ScreenToClient(Point point)
```

## Parameters

point [Point](#)

The screen coordinates to convert.

## Returns

[Point](#)

The converted client-area coordinates as a Point.

## SetHeight(int)

Sets the height of the window to the specified value.

```
void SetHeight(int height)
```

Parameters

**height** [int](#)

The new height for the window.

## SetIcon(Image)

Sets the icon of the window.

```
void SetIcon(Image image)
```

Parameters

**image** [Image](#)

The image to set as the window icon, represented as an Image of Rgba32 format.

## SetPosition(int, int)

Sets the position of the window to the specified coordinates.

```
void SetPosition(int x, int y)
```

Parameters

**x** [int](#)

The x-coordinate to set for the window position.

**y** [int](#)

The y-coordinate to set for the window position.

## SetSize(int, int)

Sets the size of the window to the specified width and height.

```
void SetSize(int width, int height)
```

### Parameters

**width** [int](#)

The new width for the window.

**height** [int](#)

The new height for the window.

## SetState(WindowState)

Sets the state of the window to the specified WindowState.

```
void SetState(WindowState state)
```

### Parameters

**state** [WindowState](#)

The desired state to set the window to, represented by the WindowState enum.

## SetTitle(string)

Sets the title of the window.

```
void SetTitle(string title)
```

### Parameters

**title** [string](#)

The new title to set for the window.

## SetWidth(int)

Sets the width of the window to the specified value.

```
void SetWidth(int width)
```

Parameters

width [int](#)

The new width for the window.

## SetX(int)

Sets the X coordinate of the window to the specified value.

```
void SetX(int x)
```

Parameters

x [int](#)

The new X coordinate for the window.

## SetY(int)

Sets the y-coordinate of the window's position.

```
void SetY(int y)
```

Parameters

y [int](#)

The new y-coordinate to set for the window.

## Events

## Closed

Occurs after the window has closed.

`event Action? Closed`

### Event Type

[Action](#)

## DragDrop

Occurs when a drag-and-drop operation is performed.

`event Action<string>? DragDrop`

### Event Type

[Action](#) <[string](#)>

## Exposed

Occurs when the window is exposed (made visible or unhidden).

`event Action? Exposed`

### Event Type

[Action](#)

## FocusGained

Occurs when the window gains focus.

`event Action? FocusGained`

## Event Type

[Action ↗](#)

## FocusLost

Occurs when the window loses focus.

`event Action? FocusLost`

## Event Type

[Action ↗](#)

## GamepadAdded

Occurs when a new gamepad is detected and added to the system. The event provides the ID of the newly connected gamepad.

`event Action<uint>? GamepadAdded`

## Event Type

[Action ↗ <uint ↗>](#)

## GamepadAxisMoved

Invoked when a gamepad's axis is moved. This event provides details such as the gamepad ID, the specific axis, and the position value of the axis movement.

`event Action<uint, GamepadAxis, short>? GamepadAxisMoved`

## Event Type

[Action ↗ <uint ↗, GamepadAxis, short ↗>](#)

## GamepadButtonDown

Occurs when a gamepad button is pressed down. The event provides the gamepad ID and the button that was pressed.

`event Action<uint, GamepadButton>? GamepadButtonDown`

### Event Type

[Action](#) <[uint](#), [GamepadButton](#)>

## GamepadButtonUp

Event triggered when a gamepad button is released. The event handler receives two parameters: the ID of the gamepad and the button that was released.

`event Action<uint, GamepadButton>? GamepadButtonUp`

### Event Type

[Action](#) <[uint](#), [GamepadButton](#)>

## GamepadRemoved

Occurs when a gamepad is removed from the system. The event provides the ID of the removed gamepad as an argument.

`event Action<uint>? GamepadRemoved`

### Event Type

[Action](#) <[uint](#)>

## Hidden

Occurs when the window is hidden.

```
event Action? Hidden
```

## Event Type

[Action](#)

## KeyDown

Occurs when a key is pressed.

```
event Action<KeyEvent>? KeyDown
```

## Event Type

[Action](#) <[KeyEvent](#)>

## KeyUp

Occurs when a key is released.

```
event Action<KeyEvent>? KeyUp
```

## Event Type

[Action](#) <[KeyEvent](#)>

## MouseButtonDown

Occurs when a mouse button is pressed.

```
event Action<MouseEvent>? MouseButtonDown
```

## Event Type

[Action](#) <[MouseEvent](#)>

## MouseButtonUp

Occurs when a mouse button is released.

**event** Action<MouseEvent>? MouseButtonUp

Event Type

[Action](#) <MouseEvent>

## MouseEntered

Occurs when the mouse enters the window.

**event** Action? MouseEntered

Event Type

[Action](#)

## MouseLeft

Occurs when the mouse leaves the window.

**event** Action? MouseLeft

Event Type

[Action](#)

## MouseMove

Occurs when the mouse is moved.

**event** Action<Vector2>? MouseMove

## Event Type

[Action](#) <[Vector2](#)>

## MouseWheel

Occurs when the mouse wheel is scrolled.

**event** Action<Vector2>? MouseWheel

## Event Type

[Action](#) <[Vector2](#)>

## Moved

Occurs when the window is moved.

**event** Action<Point>? Moved

## Event Type

[Action](#) <[Point](#)>

## Resized

Occurs when the window is resized.

**event** Action? Resized

## Event Type

[Action](#)

## Shown

Occurs when the window is shown.

`event Action? Shown`

Event Type

[Action](#)

## TextInput

Occurs when text input is received from the user. The event handler receives an array of characters representing the text that was entered.

`event Action<char[]>? TextInput`

Event Type

[Action](#) <[char](#)[]>

# Class Sdl3Window

Namespace: [Bliss.CSharp.Windowing](#)

Assembly: Bliss.dll

```
public class Sdl3Window : Disposable, IWindow, IDisposable
```

## Inheritance

[object](#) ← [Disposable](#) ← Sdl3Window

## Implements

[IWindow](#), [IDisposable](#)

## Inherited Members

[Disposable.HasDisposed](#) , [Disposable.Dispose\(\)](#) , [Disposable.ThrowIfDisposed\(\)](#) , [object.Equals\(object\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Constructors

### Sdl3Window(int, int, string, WindowState)

Initializes a new instance of the [Sdl3Window](#) class with the specified width, height, title, and window state.

```
public Sdl3Window(int width, int height, string title, WindowState state)
```

## Parameters

**width** [int](#)

The width of the window in pixels.

**height** [int](#)

The height of the window in pixels.

**title** [string](#)

The title of the window.

#### **state** [WindowState](#)

The initial state of the window, specified as a [WindowState](#) value.

## Exceptions

### [Exception](#)

Thrown if SDL fails to initialize the subsystem required for creating the window.

# Properties

## Exists

Indicates whether the window currently exists.

```
public bool Exists { get; }
```

## Property Value

### [bool](#)

## Handle

Represents the native handle of the SDL window, which is used for low-level window operations and interactions.

```
public nint Handle { get; }
```

## Property Value

### [nint](#)

## Id

Represents the unique identifier for the SDL window. This identifier can be used to reference or differentiate between multiple windows.

```
public uint Id { get; }
```

Property Value

[uint](#)

## IsFocused

Indicates whether the window is currently focused.

```
public bool IsFocused { get; }
```

Property Value

[bool](#)

## SwapchainSource

Represents the source of the swapchain, which is used for managing the rendering surface and synchronization for the window.

```
public SwapchainSource SwapchainSource { get; }
```

Property Value

[SwapchainSource](#)

## Methods

### ClearState()

Resets the window to its default state by clearing various settings such as resizability, fullscreen mode, border visibility, and always-on-top status.

```
public void ClearState()
```

## Exceptions

### [InvalidOperationException](#)

Thrown if the window handle is invalid.

## ClientToScreen(Point)

Converts a point from client-area coordinates to screen coordinates.

```
public Point ClientToScreen(Point point)
```

## Parameters

### [point](#) [Point](#)

The point in client-area coordinates to be converted.

## Returns

### [Point](#)

The point in screen coordinates.

## Dispose(bool)

Disposes of the object and releases associated resources.

```
protected override void Dispose(bool disposing)
```

## Parameters

### [disposing](#) [bool](#)

True if called from user code; false if called from a finalizer.

## GetHeight()

Retrieves the height of the window.

```
public int GetHeight()
```

Returns

[int](#)

The height of the window in pixels.

## GetOrCreateOpenGLPlatformInfo(GraphicsDeviceOptions, GraphicsBackend)

Retrieves or creates the OpenGL platform information for the current window.

```
public OpenGLPlatformInfo GetOrCreateOpenGLPlatformInfo(GraphicsDeviceOptions options, GraphicsBackend backend)
```

Parameters

[options](#) [GraphicsDeviceOptions](#)

Options for configuring the graphics device.

[backend](#) [GraphicsBackend](#)

The graphics backend to use.

Returns

[OpenGLPlatformInfo](#)

The OpenGL platform information associated with the current window.

Exceptions

[VeldridException](#)

Thrown if unable to create the OpenGL context, potentially due to insufficient system support for the requested profile, version, or swapchain format.

## GetPosition()

Retrieves the current position of the window.

```
public (int, int) GetPosition()
```

Returns

([int](#), [int](#))

A tuple containing the x and y coordinates of the window's position.

Exceptions

[Exception](#)

Thrown if there is an error retrieving the window's position.

## GetSize()

Retrieves the current width and height of the window in pixels.

```
public (int, int) GetSize()
```

Returns

([int](#), [int](#))

A tuple containing two integers representing the width and height of the window in pixels.

## GetState()

Retrieves the current state of the window.

```
public WindowState GetState()
```

Returns

### [WindowState](#)

The current state of the window represented by the [WindowState](#) enumeration.

## GetTitle()

Retrieves the title of the window.

```
public string GetTitle()
```

Returns

### [string](#)

The title of the window as a string.

## GetWidth()

Gets the width of the window.

```
public int GetWidth()
```

Returns

### [int](#)

The width of the window in pixels.

## GetX()

Retrieves the current X-coordinate position of the window.

```
public int GetX()
```

Returns

[int](#)

The X-coordinate of the window's position.

## GetY()

Retrieves the Y-coordinate of the window's position.

```
public int GetY()
```

Returns

[int](#)

The Y-coordinate of the window's position.

## HasState(WindowState)

Determines if the current window state matches the specified state.

```
public bool HasState(WindowState state)
```

Parameters

**state** [WindowState](#)

The window state to compare with the current state.

Returns

[bool](#)

True if the current window state matches the specified state; otherwise, false.

## PumpEvents()

Processes all pending events for the window and invokes corresponding event handlers.

```
public void PumpEvents()
```

### Exceptions

[Win32Exception](#)

Thrown if an error occurs when processing events.

## ScreenToClient(Point)

Converts a point from screen coordinates to client coordinates.

```
public Point ScreenToClient(Point point)
```

### Parameters

**point** [Point](#)

The point in screen coordinates to be converted.

### Returns

[Point](#)

The point in client coordinates.

## SetHeight(int)

Sets the height of the window to the specified value.

```
public void SetHeight(int height)
```

### Parameters

`height` [int](#)

The new height of the window.

## SetIcon(Image)

Sets the icon for the SDL3 window using the provided image.

```
public void SetIcon(Image image)
```

Parameters

`image` [Image](#)

The image to set as the window icon. It should be of type [Image](#).

Exceptions

[Exception](#)

Thrown if an error occurs while setting the window icon.

## SetPosition(int, int)

Sets the position of the window on the screen.

```
public void SetPosition(int x, int y)
```

Parameters

`x` [int](#)

The x-coordinate of the window position.

`y` [int](#)

The y-coordinate of the window position.

## SetSize(int, int)

Sets the size of the window to the specified width and height.

```
public void SetSize(int width, int height)
```

Parameters

**width** [int](#)

The new width of the window.

**height** [int](#)

The new height of the window.

## SetState(WindowState)

Sets the state of the window to the specified state.

```
public void SetState(WindowState state)
```

Parameters

**state** [WindowState](#)

The desired state for the window, specified as a [WindowState](#).

## SetTitle(string)

Sets the title of the window.

```
public void SetTitle(string title)
```

Parameters

**title** [string](#)

The new title to set for the window.

# Exceptions

## [InvalidOperationException](#)

Thrown if the title could not be set due to an internal error.

## setWidth(int)

Sets the width of the window.

```
public void SetWidth(int width)
```

Parameters

`width` [int](#)

The new width of the window.

## SetX(int)

Sets the X coordinate of the window's position.

```
public void SetX(int x)
```

Parameters

`x` [int](#)

The new X coordinate of the window.

## SetY(int)

Sets the Y-coordinate of the window's position.

```
public void SetY(int y)
```

Parameters

y [int](#)

The new Y-coordinate.

## Events

### Closed

Occurs after the window has closed.

```
public event Action? Closed
```

Event Type

[Action](#)

### DragDrop

Occurs when a drag-and-drop operation is performed.

```
public event Action<string>? DragDrop
```

Event Type

[Action](#) <[string](#)>

### Exposed

Occurs when the window is exposed (made visible or unhidden).

```
public event Action? Exposed
```

Event Type

[Action](#)

## FocusGained

Occurs when the window gains focus.

```
public event Action? FocusGained
```

Event Type

[Action](#)

## FocusLost

Occurs when the window loses focus.

```
public event Action? FocusLost
```

Event Type

[Action](#)

## GamepadAdded

Event triggered when a new gamepad is connected to the system.

```
public event Action<uint>? GamepadAdded
```

Event Type

[Action](#) <[uint](#)>

## GamepadAxisMoved

Represents an event that is triggered when a gamepad axis is moved.

```
public event Action<uint, GamepadAxis, short>? GamepadAxisMoved
```

## Event Type

[Action](#)<[uint](#), [GamepadAxis](#), [short](#)>

## GamepadButtonDown

Occurs when a button on the gamepad is pressed.

```
public event Action<uint, GamepadButton>? GamepadButtonDown
```

## Event Type

[Action](#)<[uint](#), [GamepadButton](#)>

## GamepadButtonUp

Triggered when a gamepad button is released, providing the button identifier and related data.

```
public event Action<uint, GamepadButton>? GamepadButtonUp
```

## Event Type

[Action](#)<[uint](#), [GamepadButton](#)>

## GamepadRemoved

Event triggered when a gamepad is removed from the system.

```
public event Action<uint>? GamepadRemoved
```

## Event Type

[Action](#)<[uint](#)>

## Hidden

Occurs when the window is hidden.

```
public event Action? Hidden
```

Event Type

[Action](#)

## KeyDown

Occurs when a key is pressed.

```
public event Action<KeyEvent>? KeyDown
```

Event Type

[Action](#) <[KeyEvent](#)>

## KeyUp

Occurs when a key is released.

```
public event Action<KeyEvent>? KeyUp
```

Event Type

[Action](#) <[KeyEvent](#)>

## MouseButtonDown

Occurs when a mouse button is pressed.

```
public event Action<MouseEvent>? MouseButtonDown
```

Event Type

[Action](#) <MouseEvent>

## MouseButtonUp

Occurs when a mouse button is released.

```
public event Action<MouseEvent>? MouseButtonUp
```

Event Type

[Action](#) <MouseEvent>

## MouseEntered

Occurs when the mouse enters the window.

```
public event Action? MouseEntered
```

Event Type

[Action](#)

## MouseLeft

Occurs when the mouse leaves the window.

```
public event Action? MouseLeft
```

Event Type

[Action](#)

## MouseMove

Occurs when the mouse is moved.

```
public event Action<Vector2>? MouseMove
```

## Event Type

[Action](#) <[Vector2](#)>

## MouseWheel

Occurs when the mouse wheel is scrolled.

```
public event Action<Vector2>? MouseWheel
```

## Event Type

[Action](#) <[Vector2](#)>

## Moved

Occurs when the window is moved.

```
public event Action<Point>? Moved
```

## Event Type

[Action](#) <[Point](#)>

## Resized

Occurs when the window is resized.

```
public event Action? Resized
```

## Event Type

[Action](#)

## SdlEvent

Represents an event that is triggered whenever an SDL event occurs within the window.

```
public event Action<SDL_Event>? SdlEvent
```

### Event Type

[Action](#)<SDL\_Event>

## Shown

Occurs when the window is shown.

```
public event Action? Shown
```

### Event Type

[Action](#)

## TextInput

Represents an event that triggers when text input is received.

```
public event Action<char[]>? TextInput
```

### Event Type

[Action](#)<[char](#)[]>

# Class Window

Namespace: [Bliss.CSharp.Windowing](#)

Assembly: Bliss.dll

```
public static class Window
```

## Inheritance

[object](#) ← Window

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) , [object.GetType\(\)](#) ,  
[object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) , [object.ToString\(\)](#)

## Methods

### CreateGraphicsDevice(IWindow, GraphicsDeviceOptions, GraphicsBackend)

Creates a graphics device for the specified window, based on the provided options and preferred backend.

```
public static GraphicsDevice CreateGraphicsDevice(IWindow window, GraphicsDeviceOptions  
options, GraphicsBackend preferredBackend)
```

#### Parameters

window [IWindow](#)

The window for which to create the graphics device.

options [GraphicsDeviceOptions](#)

Options for configuring the graphics device.

preferredBackend [GraphicsBackend](#)

The preferred graphics backend to use.

Returns

[GraphicsDevice](#)

A graphics device configured according to the specified options and preferred backend.

## CreateWindow(WindowType, int, int, string, WindowState, GraphicsDeviceOptions, GraphicsBackend, out GraphicsDevice)

Creates a new window based on the specified parameters.

```
public static IWindow CreateWindow(WindowType type, int width, int height, string title, WindowState state, GraphicsDeviceOptions options, GraphicsBackend preferredBackend, out GraphicsDevice graphicsDevice)
```

Parameters

**type** [WindowType](#)

The type of window to create.

**width** [int](#)

The width of the window.

**height** [int](#)

The height of the window.

**title** [string](#)

The title of the window.

**state** [WindowState](#)

The state of the window (e.g., maximized, minimized).

**options** [GraphicsDeviceOptions](#)

Options for configuring the graphics device.

**preferredBackend** [GraphicsBackend](#)

The preferred graphics backend to use.

## `graphicsDevice` [GraphicsDevice](#)

An output parameter that will hold the created graphics device.

Returns

### [IWindow](#)

An implementation of [IWindow](#) corresponding to the specified parameters.

## `GetPlatformDefaultBackend()`

Determines the default graphics backend for the current platform.

```
public static GraphicsBackend GetPlatformDefaultBackend()
```

Returns

### [GraphicsBackend](#)

The default [GraphicsBackend](#) for the current platform.

# Enum WindowState

Namespace: [Bliss.CSharp.Windowing](#)

Assembly: Bliss.dll

```
[Flags]
public enum WindowState
```

## Fields

**AlwaysOnTop = 8**

The window is always on top of other windows, maintaining its topmost position.

**BorderlessFullScreen = Resizable | FullScreen**

The window is borderless, removing the title bar and window borders.

**CaptureMouse = Resizable | Hidden**

The window captures the mouse, confining its movement to the window area.

**FullScreen = 2**

The window is in full-screen mode, occupying the entire screen.

**Hidden = FullScreen | Maximized**

The window is hidden and not visible to the user.

**Maximized = 4**

The window is maximized, taking up the largest possible area on the screen.

**Minimized = Resizable | Maximized**

The window is minimized, reducing it to an icon or taskbar entry.

**None = 0**

Indicates that the window has no specific state.

**Resizable = 1**

The window is resizable, allowing the user to adjust its size.

**Transparent = Resizable | AlwaysOnTop**

The window is transparent, allowing content behind it to be partially visible.

# Enum WindowType

Namespace: [Bliss.CSharp.Windowing](#)

Assembly: Bliss.dll

```
public enum WindowType
```

## Fields

Sdl3 = 0

# Namespace Bliss.CSharp.Windowing.Events

## Structs

[KeyEvent](#)

[MouseEvent](#)

# Struct KeyEvent

Namespace: [Bliss.CSharp.Windowing.Events](#)

Assembly: Bliss.dll

```
public struct KeyEvent
```

## Inherited Members

[ValueType.Equals\(object\)](#) , [ValueType.GetHashCode\(\)](#) , [ValueType.ToString\(\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### KeyEvent(KeyboardKey, bool, bool)

Initializes a new instance of the [KeyEvent](#) class with the specified keyboard key, key state, and repeat flag.

```
public KeyEvent(KeyboardKey keyboardKey, bool isDown, bool repeat)
```

## Parameters

### keyboardKey [KeyboardKey](#)

The [KeyboardKey](#) representing the key involved in the event.

### isDown [bool](#)

Indicates whether the key is pressed down ([true](#)) or released ([false](#)).

### repeat [bool](#)

Indicates whether the key event is a repeated key press ([true](#)) or a single press ([false](#)).

## Fields

### IsDown

Indicates whether the key is currently pressed down.

```
public bool IsDown
```

Field Value

[bool](#)

## KeyboardKey

Represents a key on the keyboard that can trigger key events.

```
public KeyboardKey KeyboardKey
```

Field Value

[KeyboardKey](#)

## Repeat

Indicates whether the key press event is a repeat of the previous key press action.

```
public bool Repeat
```

Field Value

[bool](#)

# Struct MouseEvent

Namespace: [Bliss.CSharp.Windowing.Events](#)

Assembly: Bliss.dll

```
public struct MouseEvent
```

## Inherited Members

[ValueType.Equals\(object\)](#) , [ValueType.GetHashCode\(\)](#) , [ValueType.ToString\(\)](#) ,  
[object.Equals\(object, object\)](#) , [object.GetType\(\)](#) , [object.ReferenceEquals\(object, object\)](#)

## Constructors

### MouseEvent(MouseButton, bool)

Initializes a new instance of the MouseEvent class with the specified button and state.

```
public MouseEvent(MouseButton button, bool isDown)
```

#### Parameters

**button** [MouseButton](#)

The mouse button that triggered the event.

**isDown** [bool](#)

A value indicating whether the button is pressed (true) or released (false).

## Fields

### Button

Represents the mouse button involved in the event.

```
public MouseButton Button
```

Field Value

[MouseButton](#)

## IsDown

Indicates whether the mouse button is pressed down.

```
public bool IsDown
```

Field Value

[bool](#) ↗