G-Raff Documentation Document

Giraffe version

# Creating a new game

Starting a new game in G-Raff is very simple. All initialization is done by calling GRaff.Giraffe.Run with the appropriate arguments. Consider the following code snippet:

1 using GRaff;

2

3 namespace MyGame

4 {

5 static class Program

6 {

7 static void Main(string[] args)

8 {

9 Giraffe.Run(1280, 800, gameStart);

10 }

11

12 static void gameStart()

13 {

14 // Initialize your game here

15 }

16 }

17 }

This will create a game window with width 1280 and height 800. After all the initialization is completed, gameStart() will be called.

DEVELOPER NOTE: Running the game will create a window and run it through a function labelled with [System.STAThreadAttribute]. The C# project should support being built as such (e.g. when creating the project, select the Windows Forms template). Furthermore, do not label Program.Main with [STAThread].

# Creating game objects

## Defining the class

G-Raff keeps track of all objects that should be drawn and updated each loop in a list called the **instance registry**. The most fundamental class in the game is GameElement. This abstract class defines two virtual methods: OnStep() and OnDraw(). Each step, G-Raff will automatically call OnStep for each existing GameElement, and each draw loop it will call OnDraw.

Note that OnDraw should not have any side effects like moving or destroying elements, as it is not guaranteed it will be called as frequently as OnStep.

The following snippet implements a GRaff.GameElement that will show a circle moving up and down:

using GRaff;

namespace MyGame

{

public class TestElement : GameElement

{

double t;

public override void OnStep()

{

t += 0.05;

}

public override void OnDraw()

{

Fill.Circle(Color.Blue, 500, 500 + 250 \* GMath.Sin(t), 50);

}

}

}

## Instantiating the class in the game room

To create an instance of a game element in the game, simply call its constructor. In the constructor of GameElement, the element will automatically add itself to the instance registry. The following snippet will instantiate a TestElement at the beginning of the game

12 static void gameStart()

13 {

14 new TestElement();

15 }

In order to create objects that should not be added to the instance registry, the recommended practice is to create a class that does not inherit GameElement. That is, each instance of a GameElement subclass is also in the instance registry. If you still need to create a game element that is not in the instance registry, you can set the property Instance.RegisterAutomatically to false.

## The GameObject class

While GameElement is the most fundamental class, it is very minimalistic. In most applications, you want to use GameObject instead. This abstract class represents a game element that exists in the room; that is, it has properties such as position, transform (scale, etc.), a sprite and a collision mask.

The GameObject also features two new virtual methods: OnBeginStep() and OnEndStep(). These provide finer control of the order in which things happen each step (see section about the game loop). For optimization purposes, GameElement does not offer these methods.

Most of the things on the screen will typically inherit from GameObject. Only things that do not have a definite location in the room should inherit directly from GameElement. For example, a GUI might always be drawn at the same position in the room, and should inherit from GameElement.

# User input

Game elements can listen to various events by implementing the appropriate listener interfaces. These interfaces are named I<event>Listener.

## Keyboard

There are three listener interfaces associated with the keyboard: IKeyListener, IKeyPressListener and IKeyReleaseListener. When a key is pressed or released, a Key Press event or a Key Release event is raised respectively. Each step while the user holds a key down, a Key event is raised. That is, the event associated with IKeyListener is raised continuously each step while a key is held, while IKeyPressListener and IKeyReleaseListener events are raised only once. Thus, for a platform game, walking left or right should probably be associated with a Key event, while jumping should come from a Key Press event.

If multiple keys are held or released at the same time, the event is raised once for each key. The key pressed is passed as an argument to the event handler. Each step, all Key events are handled first. Then, the Key Press events are handled and finally the Key Release events.

The static class Keyboard gives information about the state of the keys. This can be useful to detect if modifier keys (shift, alt, ctrl) were held down, or if you want to check for events manually in a Step event without implementing IKeyListener.

## Mouse

Mouse events work very similarly to the keyboard. There are continuous listeners, press listeners and release listeners. For each button pressed, the respective event is raised once, and the static class Mouse gives the same control, as well as the position of the cursor.

Note, however, that there are six listener interfaces associated with the mouse: IMouseListener, IMousePressListener, IMouseReleaseListener, IGlobalMouseListener, IGlobalMousePressListener and IGlobalMouseReleaseListener. The first three are local listeners: these events are raised by game objects when the mouse is clicked inside their collision mask. The global versions are called when the mouse is pressed anywhere in the room. Since GameElement does not define a collision mask, the local listeners should only be implemented by GameObject.

## Input lag

In some cases, it is important to note that the state of each input is updated only once each step. This means that some events that are detected by the OS might not occur in the game. For example, if a key is pressed and released very quickly, then that step both a Key Press and Key Release event might occur. However, since the key is not held down during the step, the Key event will not be raised.

Another example is an object following the mouse (such as a custom cursor sprite). If the game is running at 30 fps, this object will move noticeably slower than the mouse cursor on the screen.

# Collisions

## Collision masks

Game objects have a Mask defining the region of the room that the object occupies. The Mask has two important properties: A MaskShape defining the shape of the mask, and a Transform which translates this MaskShape to a polygon in the room.

GameObjects with collision masks can have Collision events with other objects. Objects that listen to Collision events should inherit ICollisionListener<TOther>, where TOther is a class that inherits from GameObject, specifying which object to check for collisions with. Instances that implement ICollisionListener<TOther> are called **colliders** and instances of TOther are called **collides**.

Each step, if the mask of a collider overlaps with the mask of an instance of a collidee, the event is raised with a reference to the collidee instance as a parameter. This happens once for each collidee.

If there are few instances of one class and many of another, which one should be used as collider and which should be the collidee? In general, there is a higher overhead of handling one collider than it is to handle one collidee. Therefore, in terms of efficiency, it is better to have one collider and many collidees. If there is one player and many enemies on the screen, it is better to let the player implement ICollisionListener<Enemy>.

On the other hand, in terms of design, having many collision handlers in one class might cause clutter. One way to avoid this is to have a parent class for all the collidees that define an abstract method for collision with the collider. This is illustrated in the following example:

class Player : GameObject, ICollisionListener<Powerup>

{

public void OnCollision(Powerup other)

{

other.Collide(this);

}

}

abstract class Powerup : GameObject

{

public abstract void Collide(Player other);

}

In this example, the player is made collider for optimization purposes. However, all the collision logic is handled by subclasses of Powerup.

DEVELOPER NOTE: Currently, collisions are in fact very slow. They will be optimized at some point in the future. Feedback on performance issues is greatly appreciated.

## Animation end

When an animated image loops, it sends an Animation End event to its owner. Since this happens in the draw loop, the event handler should not have any side effects. Relevant actions could be destroying the instance if it is purely a graphical object (like an explosion animation), or change the sprite after the current sprite has animated once (like a jumping animation).

## Global event handler

The static class GlobalEvent provides hooks for handling events without having to instantiate game elements. These events are always handled after all instances have executed their actions, with the exception of GlobalEvent.OnDrawBackground which occurs before all drawing, except the drawing done by the Background class. The following snippet adds a Key Pressed listener that will quit the game when the player presses escape:

GlobalEvent.KeyPressed += (sender, e) => {

if (e.Key == Key.Escape)

Giraffe.Quit();

};

# Drawing

The static Draw and Fill classes provide methods for drawing images and geometric shapes. The usage is self-explanatory; Draw.Sprite draws a sprite, Fill.Rectangle draws a filled rectangle.

Draw actions should only occur during the draw loop, not during the update loop. The draw engine uses a double buffer that is cleared at the beginning of the draw cycle; therefore, drawing outside of the draw loop will not have any effect.

NOTE: OnDraw should not have any side effects, like moving or destroying objects! It is not guaranteed that OnDraw will be called as frequently as OnStep. For example, if the game starts lagging, OnDraw will be called less frequently in order to ensure the game itself does not slow down, and in the game editor, OnDraw might frequently be called manually while OnStep is never called at all.

DEVELOPER NOTE: In the future, developers will be able to force redraws and buffer swaps, so that things can be drawn outside of the automatic drawing loop.

DEVELOPER NOTE: G-Raff uses OpenTK to deal with its drawing. The current version builds on OpenGL ES 3.0 (OpenTK.Graphics.ES30 namespace). Advanced users can create custom shaders using the VertexShader, FragmentShader and ShaderProgram classes, but note that these classes are in an early development stage.

# Mathematical classes

## Angle

Represents an angle. Instantiate by using Angle.Deg or Angle.Rad, depending on whether you want to specify the angle in degrees or radians. After instantiating, the value will always be between 0 and 360 degrees.

In general, to represent an angle, try using the Angle structure instead of a System.Double, even if the angle will always be in the same format (radians or degrees).

## Complex

Represents a complex number.

## GMath

An extension of System.Math, that also provides additional overloads (such as GMath.Cos(Angle) or GMath.Exp(Complex)), and a few new utility functions.

## GRandom

A static class that provides several static methods for dealing with randomness. Should be used for simple random functions when you don’t want to instantiate a System.Random.

## IntRectangle

Represents an orthogonal rectangle with integer coordinates. Internally, the struct stores the values of the x- and y-coordinates, the width, and the height.

## IntVector

Represents a 2-dimensional vector with integer coordinates.

## Line

Represents a line between two points. Internally, the struct stores the values of the two endpoins.

## Matrix

Represents an affine matrix on the form .

Vectors and Points can be transformed by multiplying them by a matrix. Instances of the Point structure are considered matrices on the form , while Vectors can be viewed as matrices on the form . This means that Vectors are not influenced by the translational part of the Matrix, i.e. the entries and .

## Point

Represents a point in space.

## Polygon

Represents a positively-oriented convex polygon.

When creating a polygon, a list of vertex points must be provided. If these vertices do not specify a convex polygon, a System.ArgumentException is thrown. If the vertices are not positively-oriented, the order is reversed before storing it internally.

## Rectangle

Represents an orthogonal rectangle. Internally, the struct stores the values of the x- and y-coordinates, the width, and the height.

## Transform

Represents an affine transform, with explicit scale, rotation, shear and translation.

## Vector

# Timing

**NB!** G-Raff supports various methods to deal with delayed and automatic actions. These classes are not thoroughly tested yet, and as such, the design might change or there could be unexpected bugs in special cases. Please use these classes as you normally would, and report any unexpected behaviour.

## Alarms

Sometimes you might want to have a delayed event. For example, if you acquire a temporary powerup, you might want it to expire after a set time, or you might want to some life at set intervals. In this case, the Alarm class might be useful.

To start a new Alarm, you can use the static Alarm.Start() method. You can also instantiate an Alarm directly, and call Alarm.Play() on that instance. Note that Alarms inherit from GameElement.

For example, for an object with a temporary lifetime, you can use the following snippet:

Alarm.Start(60, false, (sender, e) => this.Destroy());

This will destroy the instance after 60 steps. To regenerate life at set intervals, consider a snippet as follows:

regenerateAlarm = Alarm.Start(15, true, (sender, e) => this.Life += 3.5);

Note that the second parameter, isLooping, is set to true, to indicate that the Alarm should automatically repeat itself continuously. Also note that here, the created Alarm needs to be stored in a variable. If the this instance that created the alarm is destroyed, the alarm should also be removed, otherwise it will keep a reference to the instance and it will not be garbage collected:

public override void OnDestroy()

{

base.OnDestroy();

regenerateAlarm.Destroy();

}

## Tweens

Tweens (also known as easing functions) are used for smooth transitions between states. Each step, they perform an action based on the value of a transition parameter. This parameter usually ranges from 0 to 1, but not always.

The exact description of how this parameter varies is defined by the **tweening function**. This function takes an input that moves from to , and returns the value that is used for the action.

Like Alarms, tweens may also perform a special action when they finish.

Tweens inherit from GameElement. To run a Tween, you can simply instantiate the class and it will run automatically without having to store the reference anywhere. The Tween class has built-in methods that create Tweens from pre-defined tweening functions.

Example: Move an object from p1 to p2, using a sine tweening function:

Tween.Sine(60, t => Location = p1 \* t + p2 \* (1 - t));

Example: Fade in a transparent object:

Tween.Linear(15, t => Image.Alpha = t);

## Timekeeping

When dealing with custom timing, the static class Time can be useful. The time span between two events can be interpreted in two ways; either as the real world time difference, or the amount of time passed in the game.

To deal with real world time difference, Time provides the methods MachineTime() and GameTime(). MachineTime() is the number of milliseconds since the computer started, and is equivalent to System.Environment.TickCount. GameTime() returns the number of milliseconds since the game began, i.e. the time at which Giraffe.Run() was called.

Real world time difference is convenient for profiling, but should not affect things in the game directly. The method Time.LoopCount() returns the number of loops since the game started. This value is insensitive to lag so it is not fit for profiling, but it can sometimes be useful for special effects. For example, to draw a circle with a pulsating radius, one can use the following snippet:

Fill.Circle(color, Location, 30 + 5 \* GMath.Sin(Time.LoopCount() / 30.0));

By using Time.LoopCount() here, it is possible to make the parameter of GMath.Sin() vary without declaring a new variable. Furthermore, if the game is paused, this value will stop incrementing.

Finally, Time.FPS() returns the number of frames per second at which the game is actually running. This value is updated approximately once per second.

# Views

Views determine how things are mapped from draw specifications to the window. Consider an operation such as

Draw.Point(new Point(x, y), Colors.Black);

Which pixel on the screen will be affected by this operation is determined by the View. When working with views, there are several important *coordinate spaces*.

First are *Window coordinates*. These coordinates range from (0, 0) in the top left corner to (Window.Width, Window.Height), and each coordinate represents a single pixel. When dealing with Window coordinates, G-Raff uses integer coordinates.

Next are *GL coordinates*. This is OpenGL’s representation of where everything is rendered. Its coordinates range from (-1, -1) in the *lower* left corner to (1, 1). That is, (-1, 1) in GL coordinates always corresponds to (0, 0) in window coordinates. If we didn’t have Views that map between coordinate spaces, all drawing operations would have to be specified in these coordinates.

Third are *Room coordinates*. These coordinates specify a point in the room. By default, the room size is the same as the Window size, but this can be changed. Drawing operations are usually specified in Room coordinates.

Final are *Draw coordinates*. These are the coordinates that are used by draw specifications. If the Draw coordinates coincide with the Window coordinates, then the point (x, y) will be drawn exactly at the pixel (x, y) in the window.

The View specifies how points are mapped from Draw coordinates to GL coordinates.

# Asynchronous operations

The main game loop is not thread-safe. Manipulating the state of game elements outside the main thread is likely to fail quickly. However, it is often useful to run background tasks, for example while loading a file or performing web requests.

In cases like this, the AsyncOperation class is convenient for performing the asynchronous operations and synchronizing the results with the game loop. This class can be set to perform a list of tasks, where each task is marked to be performed either synchronously or asynchronously. The AsyncOperation will perform asynchronous operations in parallel on the fly, and perform synchronous operations only during the Synchronization phase each loop.

If an uncaught exception occurs during an asynchronous task, three things will happen. First, the task will abort the current Task. Then, at the beginning of the next Synchronization phase, that task will call a rollback action defined by AsyncOperation.WhenAborted(). Finally, an AsyncOperationException is thrown,

EXAMPLE: A Sprite can be loaded synchronously using Sprite.Load(). This will block the thread until the Sprite is fully loaded. For simple games, this might be an easy way to deal with things, especially for Sprites that must be loaded immediately after the game begins.

However, sometimes it is not desirable to block the thread while loading the Sprite. In cases like this, one might use Sprite.LoadAsync(), which creates a new AsyncOperation that loads the Sprite.

While the AsyncOperation is running, you can display a swirling animation or some other animation that indicates waiting. When the loading is complete, a new synchronous operation is invoked. This operation is defined through AsyncOperation.ThenSync() as follows:

mySprite.LoadAsync().ThenSync(success => if (success) this.LoadCompleted());

This will call this.LoadCompleted() after the sprite is loaded. Note that LoadCompleted() is called synchronously during the next Async phase. A second way of dealing with this is to initialize loading the Sprite asynchronously at some point, and then later ensure the Sprite has been loaded at a later point:

mySprite.LoadAsync(); // Start loading the Sprite asynchronously.  
 ...  
 mySprite.Load(); // This will block until the Sprite finishes loading.

On a fast computer, this will make the loading unnoticeable, but slow computers might freeze for a moment when Sprite.Load() is called.

A third way to do this is to make the GameObject invisible before the Sprite has been loaded:

this.Sprite.LoadAsync().ThenSync(() => this.IsVisible = true);

This will make the object invisible until the sprite becomes loaded. This can be useful for example if you load images from a camera roll or download them from online, and you want to display placeholder images temporarily while the images are loading. **NB!** Loading a Sprite like this can have side effects, such as changing the Mask of an object or start raising events for an IAnimationListener.