

Custom Project Gosu Game Tutorial

Elysia Guglielmo 102578004

Table of Contents

Intro and Setup Environment	3
Gosu Window Class and Methods	4
Initialize	4
Update	4
Draw	4
Button Down	4
Set Constants	5
Custom Data Types	6
Enumerations	6
Classes	6
Drawing Functions	8
Moving Functions	9
Detecting keyboard inputs	10
Randomly Create Objects	10
Delete Objects	11
Collisions	11
Animations	13
Load Tiles	13
Gosu.milliseconds and Modulus	14
Conclusion	16
References	16

Intro and Setup Environment

The Ruby gem Gosu is a graphics library that allows you to create 2D games. In this tutorial you will learn how to create a sprite-based game with a player, enemies, collisions and animations. Keep in mind this will be done using Structured Programming as opposed to Object-Oriented Programming.

Gosu forces you to separate drawing functions and moving functions, or other functions that require updates which results in good design and abstraction. Gosu also provides classes for colour, font, image, and sound which are useful in game development.

Once you have installed the Gosu gem, create a folder for your project. In this folder there will be the ruby file containing code for the game, a text file with information about the levels, and a subfolder called 'media'. This subfolder is where images and audio for the game will be filed.

Open up your favourite IDE and create a new file. It can be named anything, but for the purposes of this project it will be `asteroid_dodge.rb`. Before continuing, you must require the necessary Gosu gems at the top of the file to get the functionality needed to create the game. These are `rubygems` and `gosu`.

```
require 'rubygems'  
require 'gosu'
```

Gosu Window Class and Methods

Before planning the game, it is important to have an understanding of how the Gosu Window works, what it looks like and what its methods do. The code snippet below shows what the Gosu gem looks like and what it provides.

Gosu is an OOP (Object-Oriented Library) and therefore uses the OOP concept Inheritance. Your game will inherit the properties and methods of the Gosu::Window class. When defining the game class, it should be the name of the game in CamelCase.

```
class SpaceShooterGame < Gosu::Window
  def initialize

  end

  def update

  end

  def draw

  end

  def button_down(id)

  end
end

SpaceShooterGame.new.show
```

Initialize

The initialise method is where the game is set up and instance variables which need to be accessible throughout the whole Gosu Window are declared. Here is an appropriate place to read in files, set up sprites and images and setup the initial game state.

Update

The update method runs continuously like a cycle. Anything you put in it will be repeatedly called. Ideally, in the update method you'd want to put any updates on movement, animations or updates on game status.

Draw

Without the draw method, the game would not actually be a GUI (Graphical User Interface). The draw method works similarly to the update method in that it is a cycle, but this method's role is to actually display, or draw the graphics. If drawing functions are put in the update method, they will not execute.

Button Down

This method responds to user input from a keyboard or mouse. It is not a cycle and its body will only execute when a button is pressed. Different actions can be applied to different buttons by accessing the button id.

Set Constants

Returning to the top of the file just below the requires, now is a good time to define constants for the game, specifically the width and height of the game window screen. Integers are interpreted by Gosu as pixels.

```
WIDTH = 1000  
HEIGHT = 800
```

These can be set in the Gosu Window in initialise. To complete the basic setup of the Gosu Window, set the caption to be the name of your game. This will show in the top bar of the window.



```
class SpaceShooterGame < Gosu::Window  
  def initialize  
    super WIDTH, HEIGHT  
    self.caption = "Asteroid Dodge"  
  
  end
```

If you run the game now, a blank screen will appear with the title of your game at the top of the window.

Custom Data Types

Enumerations

A module for Z indexes is useful to have. Z indexes are numbers which define how objects in the game should layer and overlap. The higher the Z index, the more an object will be in front. BACKGROUND will be used for the background image, MIDDLE for objects the player interacts with, PLAYER for the player, and UI for text such as the lives, score and level.

```
module ZOrder
  BACKGROUND, MIDDLE, PLAYER, UI = *0..3
end
```

Classes

Now it is time to figure out what custom data types you are going to need for the game. The best way to approach this is to think, what objects will my game have? Then the properties for the objects can be decided on. What describes the object? At a minimum, it has an image to be drawn as and it has a position in the Gosu Window. Thus, a player could have an image, x position and y position. Another property a player object would be likely to have is a score.

Gosu provides an Image class which will be used to assign images as properties for each object that requires an image. The syntax to create a new image is shown below. The string within the brackets is the image's file path relative to the ruby game file. Since sprites and images are kept in the media folder, the image must be accessed with "media/" and then the filename of the image.

```
image = Gosu::Image.new("media/spaceship.png")
```

To write this in code, a Ruby class is used as shown in the code snippet below. You can decide which properties should have predefined values when initialised or if some of the values should be customised and defined when initialised.

```
class Player
  attr_accessor :score, :image, :x, :y

  def initialize(x, y)
    @score = 0
    @image = Gosu::Image.new("media/spaceship.png")
    @x = x
    @y = y
  end
end
```

The asteroid custom data type can be made similarly, however it will require a speed attribute. `rand()` is a useful Ruby built-in function that returns a number within a specified range. By using this, the asteroids can alter in speed. When an asteroid is initialized, a random speed between 5 to 10 will be allocated to it.

```
class Asteroid
  attr_accessor :image, :x, :y, :speed

  def initialize(x, y)
    @image = Gosu::Image.new("media/asteroid.png")
    @x = x
    @y = y
    @speed = rand(5..10)
  end
end
```

Custom data types will then be instantiated in Gosu initialise which will become instance variables. Instance variables are accessible from anywhere in the scope of the Gosu Window and will later be passed in drawing, moving and collision functions. Instance variables, or Gosu Class variables, are typed with an @ in front.

Since there will be multiple asteroids, an array will be defined to store them.

```
class SpaceShooterGame < Gosu::Window
  def initialize
    super WIDTH, HEIGHT
    self.caption = "Space Shooter"

    # Instantiate player
    @player = Player.new(WIDTH/2, HEIGHT - 200)

    # Asteroids array
    @asteroids = Array.new()

    # Background image
    @background = Gosu::Image.new("media/background.png")
  end
end
```

Drawing Functions

Drawing functions are created to draw an image at a given coordinate and are called in the Gosu draw method. Below demonstrates how to use the Gosu draw function which requires an image. It will accept a Gosu Window instance variable which will be the player. The Gosu::Image type has a draw method and its syntax is `image.draw(x, y, z)`. Earlier the ZOrder module was defined and will be used here to set the z-index of the player as shown below.

```
def draw_player player
  player.image.draw(player.x, player.y, ZOrder::PLAYER)
end
```

Similarly, a drawing function can be created for the asteroids which is called to draw one at a time.

```
def draw_asteroid asteroid
  asteroid.image.draw(asteroid.x, asteroid.y, ZOrder::PLAYER)
end
```

To actually draw the object in the Gosu window, it must be called in the Gosu draw method and the player instance variable should be passed as the parameter. As an additional step, the background image can also be drawn.

```
def draw
  @background.draw(0, 0, ZOrder::BACKGROUND)
  draw_player(@player)
end
```

Now when the game is run, the player will show on the screen! The asteroids cannot be drawn yet because they need to be first pushed to the array at random and since their starting position is above the screen, they will only be visible when movement is applied. Once the move function is called for the asteroids, then the coordinates for them will update in draw and they will be drawn at every time-step.

Moving Functions

Now it's time to "get things moving"! Moving functions are used to update the positions of objects. For the player, these will be triggered by keyboard input, whereas movement for asteroids will happen automatically. To allow for the player to move horizontally in the game, functions to move the player left and right need to be created. In these functions a number will define how fast the player moves in a particular direction. This works because these functions will be called in the update Gosu function which is a cycle and continuously updates. These functions can be adjusted with different numbers to get the right feel for speed and movement.

It is important to understand how positions and graphs work in Gosu. The x axis works the same way as a regular graph you'd be familiar with in math and calculus, however, y works in the opposite way. The further downwards on the y axis, the higher the y value. The figure to the right helps to depict this. X and Y are both zero at the top left corner of the Gosu Window.

Gosu window

x++



With this knowledge, it is easy to understand how movement works in Gosu and can be applied to the move functions for our objects.

Moving left is in the negative x direction, right in the positive x direction, up in the negative y direction and down in the positive y direction. There needs to be logic to detect if the player's position is on the edge of the window. For example in move_left, "if the player has hit the left wall, keep the player's position at the left wall." The code snippet below shows an example for moving left which updates the player's x position.

```
def move_left player
  player.x -= 6.0
  if player.x <= 0
    player.x = 0
  end
end
```

A move function can be made for the asteroids also. Since the asteroids only move down, they only require one move function.

```
def move_asteroid asteroid
  asteroid.y += asteroid.speed
end
```

These functions are then called in the Gosu update method. Every time the position properties of an object is changed in the update method, the draw method will draw the object at the new position to reflect that change.

Detecting keyboard inputs

In order to be able to respond to user interactions, button presses need to be detected with use of key mappings. The Gosu method `button_down?` accepts a key from an enumeration of keys defined by Gosu and returns true or false whether that key was used. The table below lists the basic keys required for the game.

Left	Gosu::KB_LEFT
Right	Gosu::KB_RIGHT
Return	Gosu::KB_RETURN

When the left key is pressed, the player should move left, thus in the body of the if statement, `move_left` can be called with the player instance passed. This should be placed in the Gosu update method because the code to move the player needs to be updated for every frame. This is how the player will move 6 pixels per time step. The same process can be applied with moving right.

```
if Gosu.button_down? Gosu::KB_LEFT
  move_left @player
end

if Gosu.button_down? Gosu::KB_RIGHT
  move_right @player
end
```

Randomly Create Objects

As there will be multiple asteroids moving at the same time, the move function needs to be applied to all of the asteroids in the array. Before any functions can be applied, the asteroids must first randomly generate. This is done using `rand()` which accepts an integer or a range between two integers and returns a random integer within the given range. This is very useful for generating objects in games or changing behaviour in objects that are controlled by the program. Then, the push method is used to append items to the array.

```
if rand(100) == 1
  @asteroids.push(Asteroid.new())
end
```

Now the Gosu draw method can draw the asteroids by using the “each” list operation. For each of the asteroids in the array, it will be drawn to the Gosu window.

```
def draw
  @background.draw(0, 0, ZOrder::BACKGROUND)
  draw_player(@player)
  @asteroids.each { |asteroid| draw_asteroid(asteroid) }
end
```

Delete Objects

Similarly to the issue where the player was able to move off the screen, the asteroids do the same. There are many instances of asteroids created where the ones that move off the screen are left to float forever which will make the performance incredibly slow. A function can be made to remove the asteroids that are no longer needed in the game.

This is when the “reject” list operation comes in handy. It will delete items from an array if they meet a condition. In Asteroid Dodge, the asteroids are no longer useful once they pass the bottom of the screen. A function like this can be written in the Gosu Window where it directly references the asteroids instance variable and then be called in Gosu update. It's code is shown in the code snippet below.

```
def remove_asteroids
  @asteroids.reject! do |asteroid|
    if asteroid.y > HEIGHT
      true
    else
      false
    end
  end
end
```

```
def update
  # Other code...
  self.remove_asteroids
end
```

Collisions

Now when the game is run, asteroids will move down randomly. However, when the player comes in contact with one, nothing happens... yet. There is another condition for when the asteroids should be removed and that is when it hits a player.

Detecting collisions for when the player hits the edges of the screen is fairly straightforward, but how about when the player collides with another object? In this case, the solution is to use Gosu.distance which compares the x and y positions between two entities and returns true if they are within the specified range, or radius. Its parameters are entity1.x, entity1.y, entity2.x, and entity2.y. In this example, 100 is used as the distance, but this will vary depending on the size of the sprites, therefore the value should be tested to ensure the collision is accurate because a collision should not occur if there's still space between the two objects.

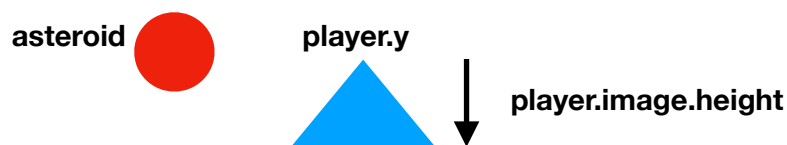
```
if Gosu.distance(@asteroid.x, @asteroid.y, @player.x, @player.y) < 100
  @player.lives -= 1
end
```

This if statement can be placed in the `remove_asteroids` function as another rejection condition. Make sure that the player lives update before returning true. The function can be rewritten as shown in the following code snippet.

```
def remove_asteroids
  @asteroids.reject! do |asteroid|
    if asteroid.y > HEIGHT
      true
    elsif Gosu.distance(@asteroid.x, @asteroid.y, @player.x, @player.y) < 100
      @player.lives -= 1
      true
    else
      false
    end
  end
end
```

When the player passes an asteroid without getting hit, it wins a point. This isn't really a collision, however, it still requires the game to detect positions between two objects. A function to update the player's score can be created. It will accept a single asteroid and the player as parameters passed by reference.

The Gosu image class has two read only properties which give the width and height of an image. Having the height of the image is necessary to detect if the player has passed the asteroid because the player y position is the top of the image. Therefore, to update the score the asteroid must pass the player y position plus the height of the image. Below is a diagram to represent this.



```
def update_player_score(asteroid, player)
  if asteroid.y > player.y + player.image.height
    player.score += 1
  end
end
```

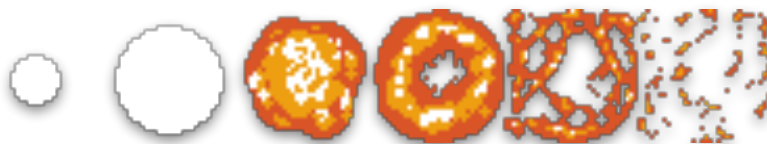
This function should be called for each asteroid. To accomplish this, the “each” list operation will be used again, but this time in the Gosu update method.

```
@asteroids.each { |asteroid| update_player_score(asteroid, @player) }
```

Animations

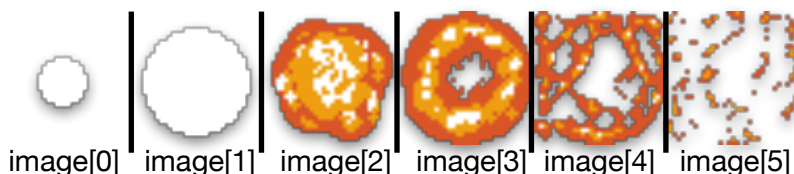
Load Tiles

An animation sprite can be used to create explosions for when the spaceship collides with an asteroid. An ideal sprite for this would have multiple frames of the animation in a single image. Each frame has equal dimensions. In the example below, each frame is 32x32 pixels.



Explosion (Sogomn 2011)

The `load_tiles` method for the Gosu Image class accepts a relative file path for the sprite and its dimensions. It then returns an array of the images it was able to 'cut out' from the sprite.



```
@image = Gosu::Image.load_tiles("media/explosion.png", 32, 32)
```

An explosion class is necessary to have in order to store the current index of the explosion image array and a property to track when the animation is complete. Having these stored in a reference to a class means that when they are updated they won't be affected by the Gosu update method constantly updating `Gosu.milliseconds`. Once new values are assigned, they are fixed. The `@index` can be zero and `@finished` can be set to false by default.

Gosu has a `Song` type which plays audio files. The explosion should have a sound effect. Similarly to the Gosu Image type, it requires a relative file path of the file to play.

```
class Explosion
  attr_accessor :image, :sound, :index, :finished, :x, :y



  def initialize(x, y)
    @x = x
    @y = y

    @image = Gosu::Image.load_tiles("media/explosion.png", 32, 32)
    @sound = Gosu::Song.new("media/atari_boom.wav")
    @index = 0
    @finished = false
  end
end
```

Gosu.milliseconds and Modulus

Milliseconds is a Gosu property which returns how much time has elapsed since the Gosu Window started running. Using Gosu.milliseconds is necessary to run code within a set time or to track time. Each tile in the array needs to be drawn in order, one after another, to run the explosion animation. Initially one might think to use a while loop to iterate through the tiles, but there is an issue that comes with that. The explosion animation will run so quickly that it can be barely visible! This is because it's drawing a new frame every millisecond, or every Gosu Window time step, which is way too fast to make the animation visible. Therefore, there needs to be a space of milliseconds between each image tile drawn. A way to work around this is to use the modulus operator.

Modulus (%) gives the remainder of two numbers divided. E.g. $20 \% 8 = 4$. This is a way of customising the frame rate for an animation. Every time $\text{Gosu.milliseconds} / \text{rate} \% \text{image.length}$ results in a number that is one of the image tile indexes, then that image tile will be drawn to the screen. A table is presented below to illustrate how it works. Note, these are only rough calculations and estimates.

<p>$1000 \text{ milliseconds} / 120 \% 6 = 2.3$ (rounds up to 3)</p> <p>The milliseconds will continue to round up to 3 for some time above 1000 milliseconds before iterating to the next tile. 120 is used as the rate, but this is experimental and should be tested with different numbers to produce the right effect.</p>	<p>image[3]</p> 
<p>$2000 \text{ milliseconds} / 120 \% 6 = 4.6$ (rounds up to 5)</p> <p>At 2000 of Gosu milliseconds the tile with index 4 shows. It can be <i>assumed</i> that the tile changes every half a second with the given rate of 120 and 6 tiles, however this is inaccurate with only testing two Gosu millisecond times (the animation is likely to be faster than this).</p>	<p>image[5]</p> 

Finally, there needs to be a check to see if the last tile has been drawn. In that case, the animation is complete and it's finished property should be set to true.

```
def draw_explosion explosion
  current_img = explosion.image[Gosu.milliseconds / 120 %
explosion.image.length]
  current_img.draw(explosion.x, explosion.y, 5, 5.0, 5.0)

  if current_img == explosion.image[explosion.image.length - 1]
    explosion.finished = true
  end
end
```

The explosions are triggered when a player crashes. Start by declaring an array in Gosu initialize for the explosions. Then, a new explosion should be pushed to the array when the collision is detected. The collision condition in `remove_asteroids` can be updated to look like this.

```
elsif Gosu.distance(@asteroid.x, @asteroid.y, @player.x, @player.y) < 100
  @explosions.push(Explosion.new())
  @player.lives -= 1
  true
else
  false
end
```

Then, set the sound effect to play for each explosion in the array. To play a song, use `play(false)`. False means that the song won't loop. Once again, the `each` list operation can be applied here and called in Gosu update.

```
@explosion.each { |explosion| explosion.sound.play(false) }
```

Now animations can be removed once they are complete. It can be created and called the same way as rejecting asteroids. An example of the rejection loop is shown.

```
def remove_explosions
  @explosions.reject! do |explosion|
    if explosion.finished
      true
    else
      false
    end
  end
end
```

If you wish to include levels, using Gosu milliseconds can also help manage pauses between levels. Every time the player completes a level, a start time property of a level class would be updated with the current Gosu milliseconds. Then, while Gosu milliseconds is within the start time and a specified time, the game will be in pause mode. Once the specified time has elapsed, the pausing stops. The code snippet below shows how to set a 6 second pause and how pauses are tracked with a boolean instance variable.

```
if Gosu.milliseconds < @levels[@level_count].start_time + 6000
  @paused = true
else
  @paused = false
end
```

Conclusion

This tutorial covered the use of the Gosu window and its methods, creating custom data types for games, drawing and moving objects, detecting collisions, working with images, animations and sound, and managing object creation or deletion throughout the game cycle. Insight was also given to what functions can be created which utilise abstraction keeping movement, drawing and collisions separate and ensuring readable code. This was taught in the context of creating the Asteroid Dodge game, but with this knowledge one can apply the techniques to any game idea one has.

References

'Explosion'[image], in Sogomn 2017, *Explosion* | [OpenGameArt.org](https://opengameart.org/sites/default/files/explosion_11.png), [OpenGameArt.org](https://opengameart.org/sites/default/files/explosion_11.png), viewed 15 June 2020, <https://opengameart.org/sites/default/files/explosion_11.png>.

jlNr, 2008, 'Animations and modulo [from wiki]', *Gosu*, 9 November, viewed 13 June 2020, <https://www.libgosu.org/cgi-bin/mwf/topic_show.pl?tid=10>

Ruby Forum, 2011, *Why can a floating point number be used as an array index?*, Ruby Forum, viewed 15 June 2020, <<https://www.ruby-forum.com/t/why-can-a-floating-point-number-be-used-as-an-array-index/205101>>.