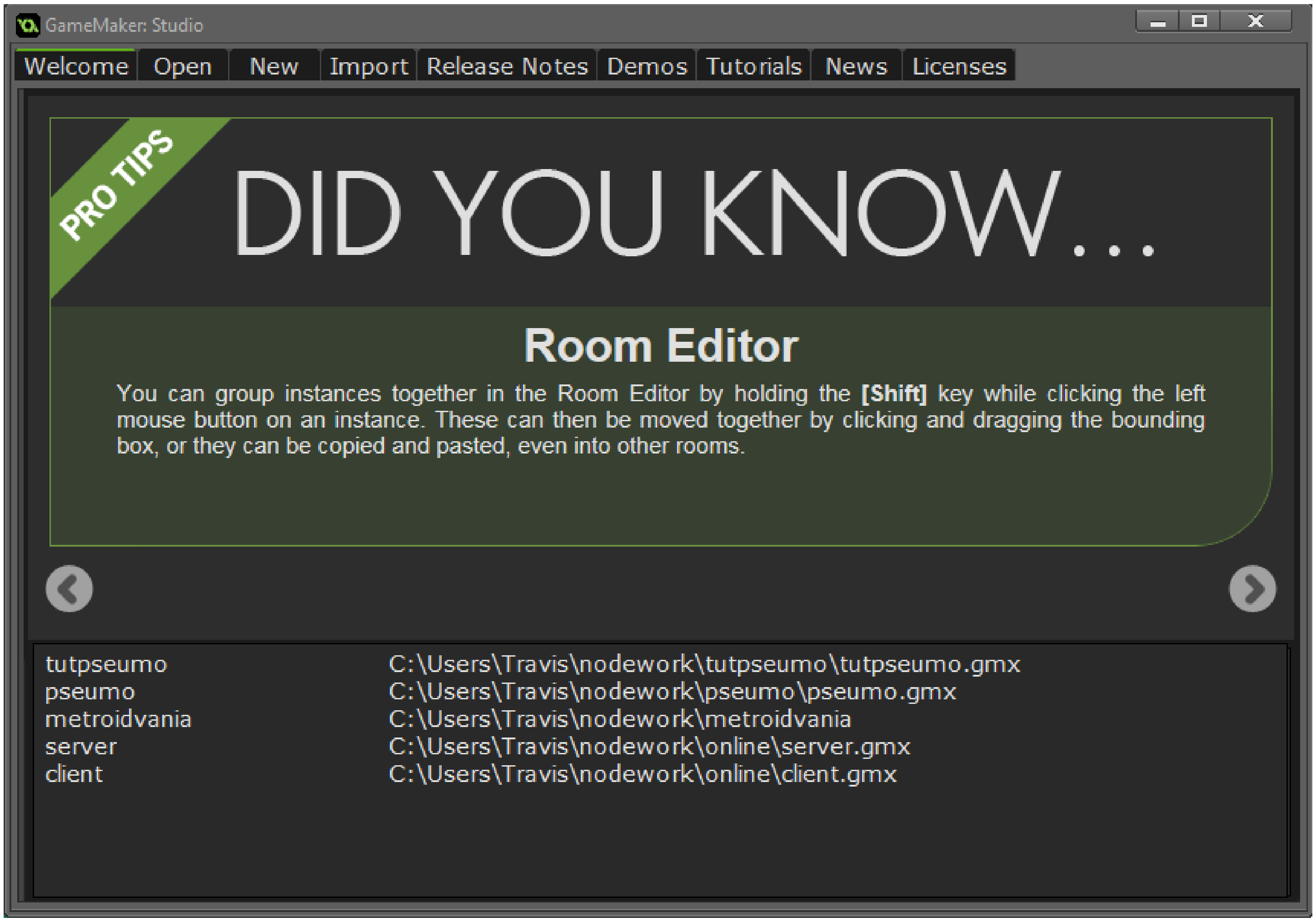
GameMaker Studio Tutorial
Physics and Movement Engine

Introduction
Game development is broad, multifaceted field with growing significance. It's difficult to know where to start when beginning to learn a new environment. GameMaker Studio (hereby: GMS) is one of my favorite environments to build 2D games in as it contains a sufficient physics system and clean asset management. GMS uses its own coding language built on C++ which uses common syntax if you're familiar, but that will not be necessary for this tutorial's content. This tutorial is intended for anyone new to either GMS or looking to begin using its built-in physics system as opposed to building your own (which is common in GMS projects). GMS has a free version available on their site (http://www.yoyogames.com/gamemaker).
The assets and the project itself are available for free on a public github repositiory (https://github.com/apoptis/tutpseumo); you're welcome to use the assets for the purpose of this tutorial but I enourage you to try and create your own assets to become more familiar with the the process. There is also a working version of the end product (https://apoptis.github.io/tutpseumo/).



Room Editor

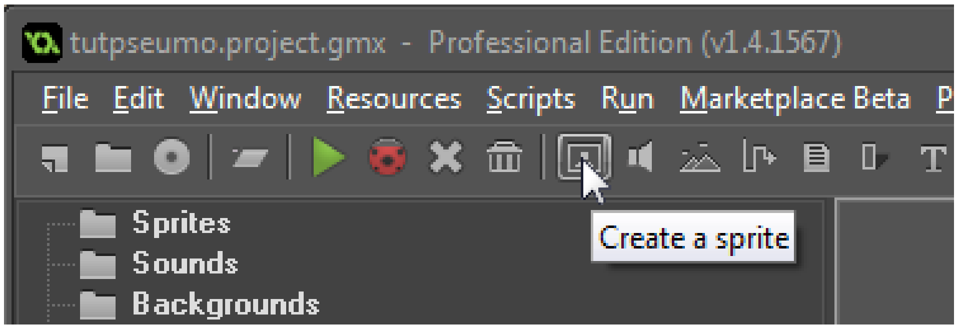
You can group instances together in the Room Editor by holding the **[Shift]** key while clicking the left mouse button on an instance. These can then be moved together by clicking and dragging the bounding box, or they can be copied and pasted, even into other rooms.

```
tutpseumo       C:\Users\Travis\nodework\tutpseumo\tutpseumo.gmx
pseumo          C:\Users\Travis\nodework\pseumo\pseumo.gmx
metroidvania    C:\Users\Travis\nodework\metroidvania
server          C:\Users\Travis\nodework\online\server.gmx
client          C:\Users\Travis\nodework\online\client.gmx
```

Upon running GameMaker Studio, you will be greeted with this screen that gives you several options. We will be using the "New" tab at the top to start our project. There will be a prompt to name your project and select it's directory path.

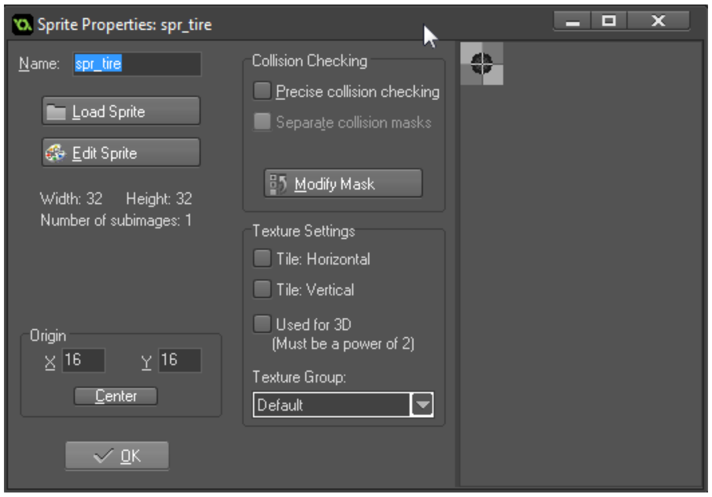
Creating the Sprites and Background

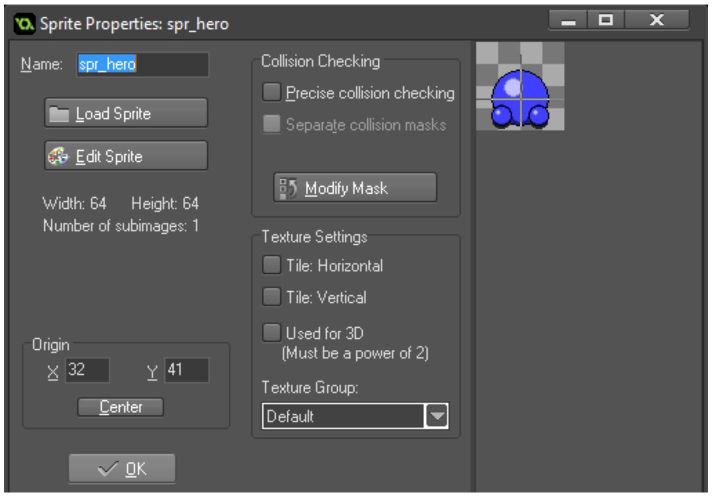
There are several places where it's possible to start but I find starting with the initial sprites that will be used helps to begin with an image of the objects that will later interact with one another. For the purposes of this tutorial we will only focus on the controllable avatar (hereby: hero) and the stage; the background will be added after using a different method.

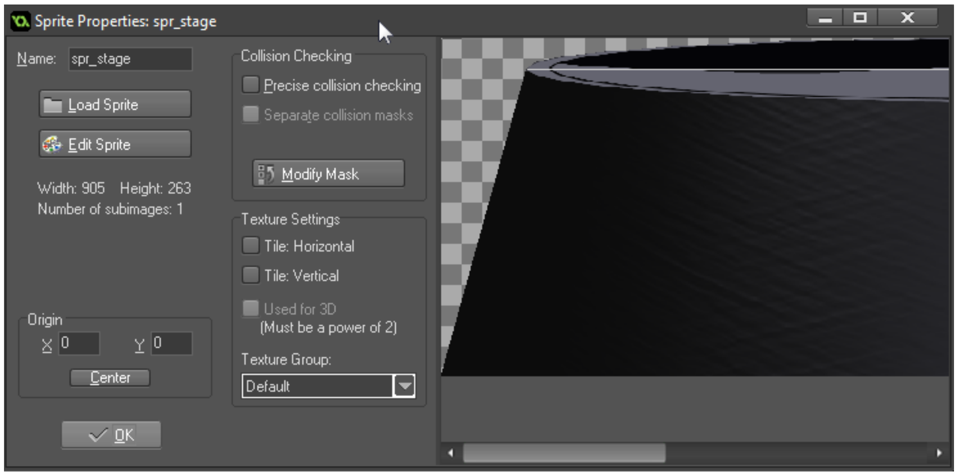

tutpseumo.project.gmx - Professional Edition (v1.4.1567)

First we will click the button at the top for creating a new sprite. This will open a new window where the sprite can be created from a file or created using GMS's built-in sprite editor (I generally use photoshop or illustrator instead but there are worse editors than GMS's).

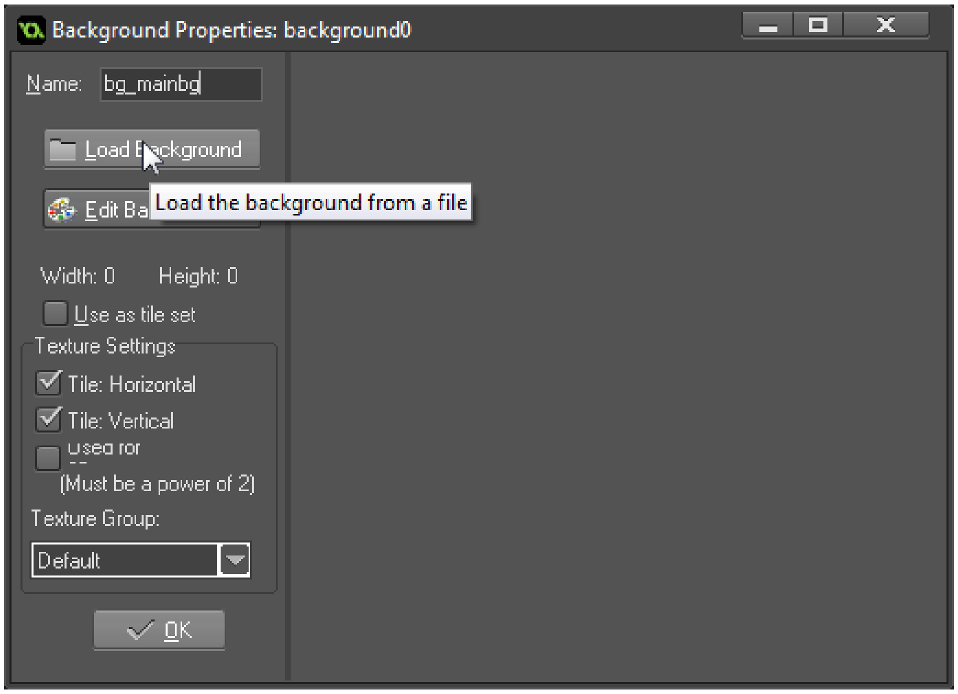
We will name the sprite "spr_hero" and either create a new sprite or load a preexisting sprite (the project files that I am using here are availabe on the public gihub project linked above). The origin is set to be centered horizontally at 32 and vertically at 47. We will use the same method for the tire's sprite but with a centered origin at 16 for both X and Y.


Sprite Properties: spr_tire
Name: spr_tire
Width: 32 Height: 32
Number of subimages: 1
Origin X 16 Y 16
Texture Group: Default


Sprite Properties: spr_hero
Name: spr_hero
Width: 64 Height: 64
Number of subimages: 1
Origin X 32 Y 41
Texture Group: Default


Sprite Properties: spr_stage
Name: spr_stage
Width: 905 Height: 263
Number of subimages: 1
Origin X 0 Y 0
Texture Group: Default

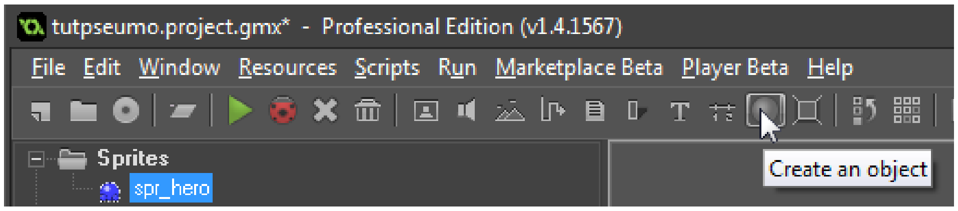
The stage sprite will created similarly but this time leaving the origin at 0 for both X and Y.



Adding a background is just as easy. The button is two to the left of the sprite button or may also be added by right clicking the background directory on the left and selecting “Create Background”. Again, you may either create a new background or use my background included in the public repository.

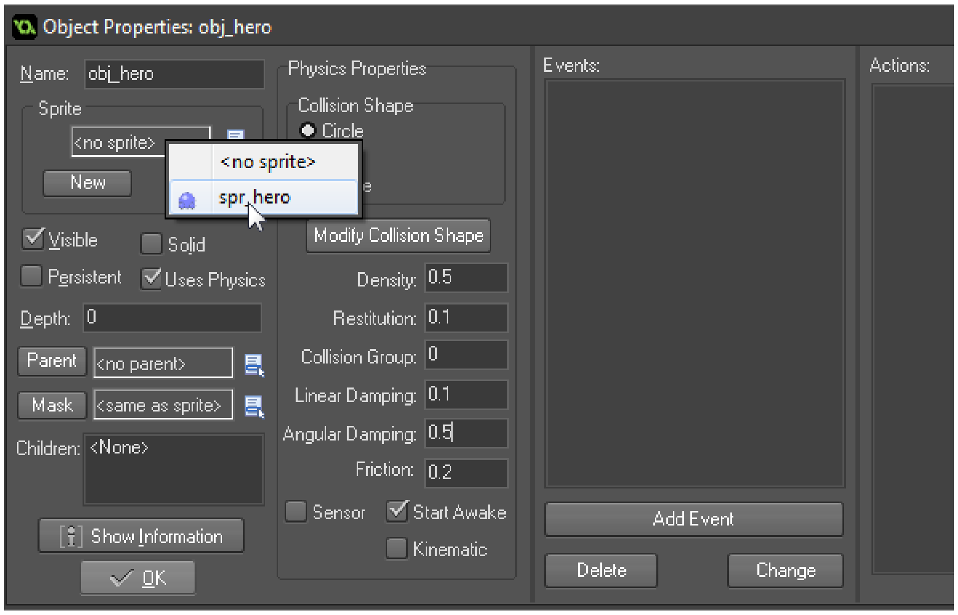
Creating the Hero, Stage, and Collision Objects

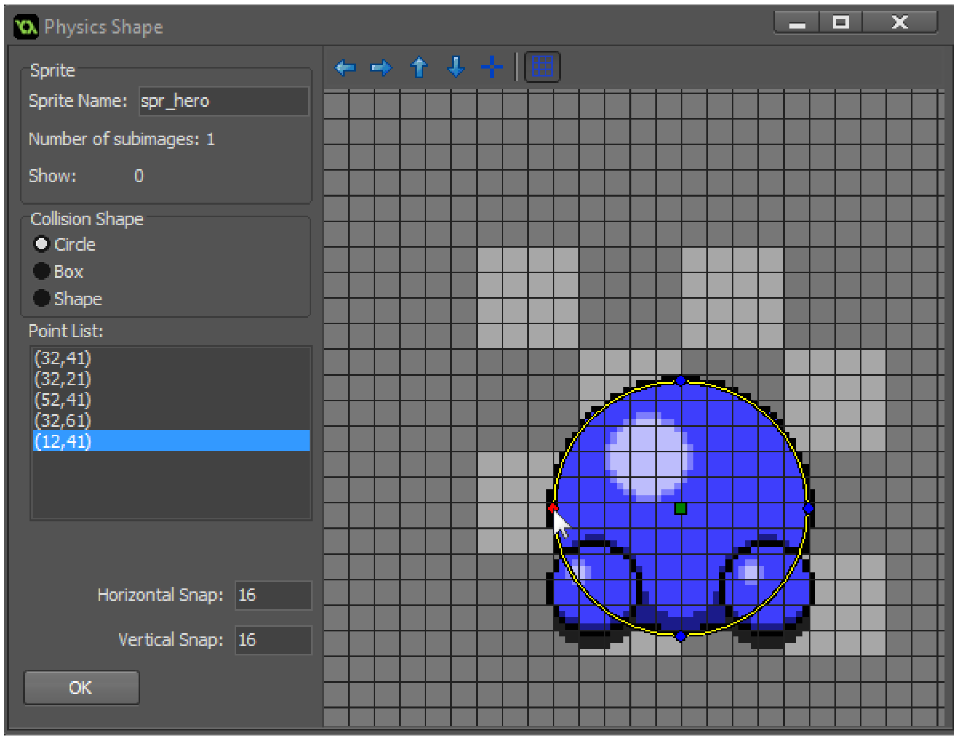
The sprite takes care of the image but we still need to create the actual objects that hold the game logic. Generally GMS projects will have several subdirectories containing several objects each; however for this tutorial we will only need 3 objects: the hero, the stage and a third object that will act as a parent (an object that passes down its properties to its children) for each of these that will allow for collisions.


tutpseumo.project.gmx* - Professional Edition (v1.4.1567)
File Edit Window Resources Scripts Run Marketplace Beta Player Beta Help
Sprites
spr_hero
Create an object

Clicking the “Create an object” button at the top will open up the settings for a new object.

From this window, we will select the sprite for the hero and also click the checkbox for “Uses Physics” which will allow for the object to take advantage of GMS’s built-in physics system. Enabling phyics will create a few other options which I’ve changed but I recommend tinkering with the numbers to see how they affect the object’s behavior. This should be repeated with the stage as well with the exception of setting the “Density” variable to 0 so that it will become stationary.




Physics Shape

Sprite

Sprite Name: spr_hero

Number of subimages: 1

Show: 0

Collision Shape

[✓ Circle] Box Shape

Point List:

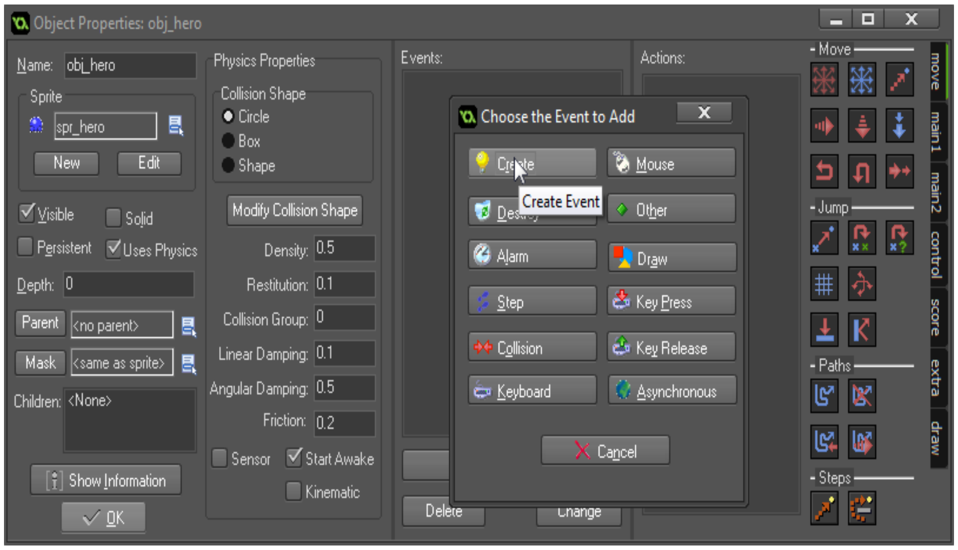
(32,41)
(32,21)
(52,41)
(32,61)
(12,41)

Horizontal Snap: 16

Vertical Snap: 16

Click “Modify Collision Shape” to be able to change the shape of the area of the object that will collide with other solid objects. This must be done on both the hero and the tires in order to avoid awkward displacement above the stage.

Now we will add our first event for the hero object. Click “Add Event” and then click “Create” to add an event that will run each time the hero object is created. This is how we will intialize all of the variables that are necessary for the object to operate.


Object Properties: obj_hero

Name: obj_hero

Sprite: spr_hero

[✓ Visible] Solid

Persistent [✓ Uses Physics]

Depth: 0

Parent: <no parent>

Mask: <same as sprite>

Children: <None>

Physics Properties

Collision Shape: [✓ Circle] Box Shape

Density: 0.5

Restitution: 0.1

Collision Group: 0

Linear Damping: 0.1

Angular Damping: 0.5

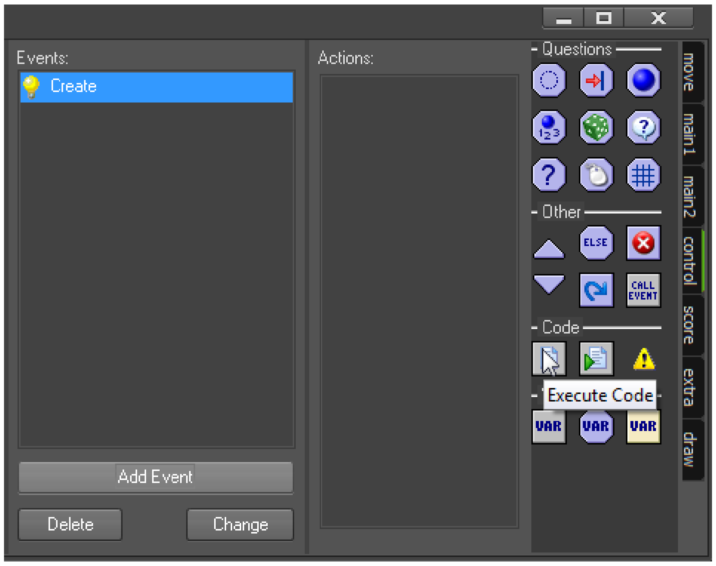
Friction: 0.2

Sensor [✓ Start Awake] Kinematic

Events:

Actions:

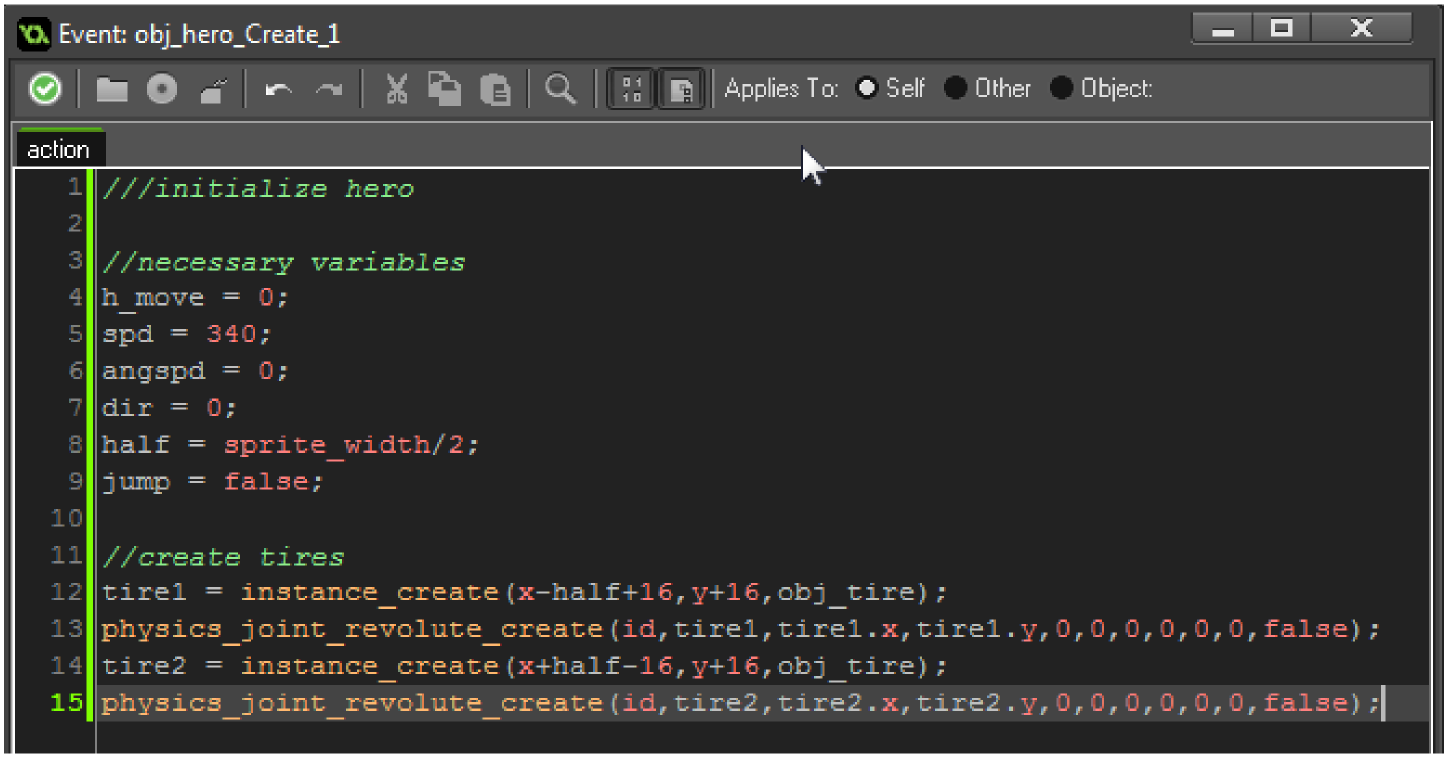
Choose the Event to Add


Events:

Create

Actions:

After adding the create event, we will need to add an action to run upon creation. In the tabs on the right, there is a tab named “control”; here we will find a button the execute code. This is generally the most used action in GMS projects as it allows for the most freedom of all the actions (including being able to use every other action if coded to do so).

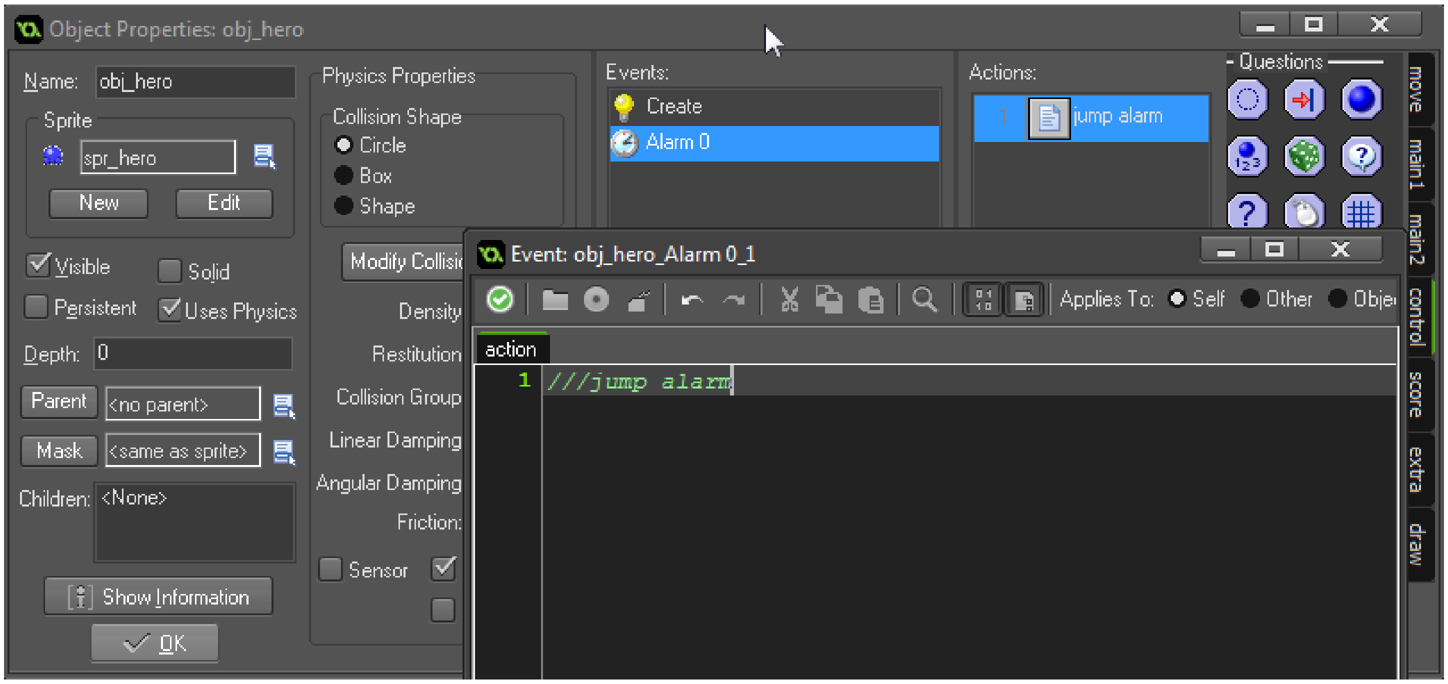


```
///initialize hero

//necessary variables
h_move = 0;
spd = 340;
angspd = 0;
dir = 0;
half = sprite_width/2;
jump = false;

//create tires
tire1 = instance_create(x-half+16,y+16,obj_tire);
physics_joint_revolute_create(id,tire1,tire1.x,tire1.y,0,0,0,0,0,0,false);
tire2 = instance_create(x+half-16,y+16,obj_tire);
physics_joint_revolute_create(id,tire2,tire2.x,tire2.y,0,0,0,0,0,0,false);
```

The above code is mostly simple in that its main purpose is to initialize variables attributed to the hero object. "h_move" is the variable for horizontal movement; "spd" is the variable for the speed that we will rotate the object; "dir" is the direction the hero will spin; "half" is the point halfway on the hero horizontally; "jump" is a boolean value which means it can only be true or false. The code beneath that is creating the tire objects and using the "physics_join_revolute_create(...)" code to append them to the hero so that they behave as wheels.

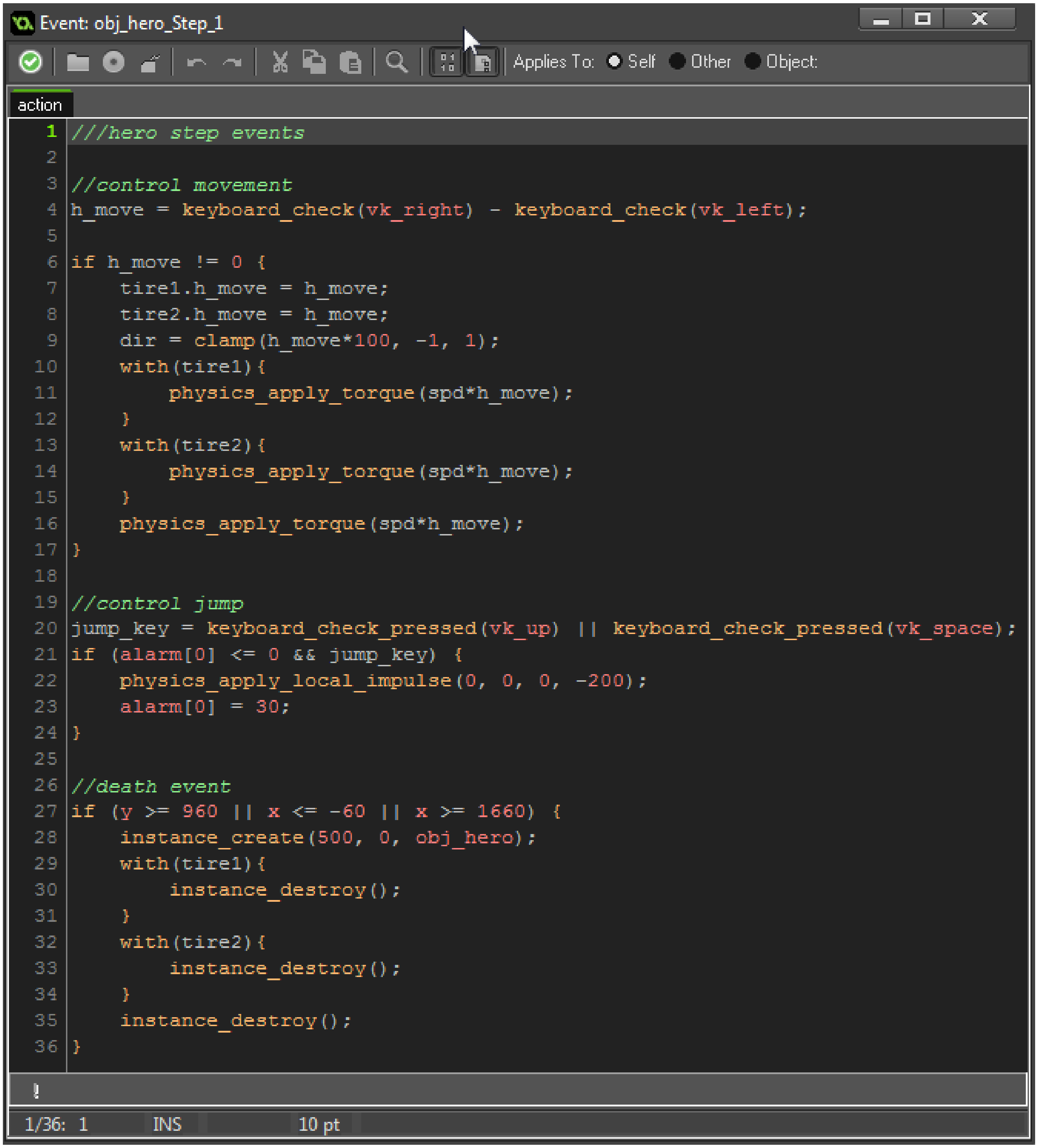


Name: obj_hero
Sprite: spr_hero
Collision Shape: [✓ Circle] Box Shape
[✓ Visible] Solid
Persistent [✓ Uses Physics]
Depth: 0
Parent: <no parent>
Mask: <same as sprite>
Children: <None>
Events: Create, Alarm 0
Actions: 1 jump alarm

```
///jump alarm
```

An alarm event will also be added, along with the code action that only contains the line "///jump alarm" which will make the upcoming code for the jump event only be able to run once the time on the alarm has run out.

We will then create a “Step” event just like we added the “Create” event that will run every “step” which means each time the game screen refreshes (several times a second; the default is 30). The code below is commented well enough and mostly understandable but some things may need some more explanation. The “keyboard_check(...)” calls return a value of 0 or 1 if the key (vk_right or vk_left) are pressed. The block beneath that checks to see if h_move does not equal 0 and if not, sets the angular movement of both the hero and its tires. The jump event is similar though it also uses “physics_apply_local_impulse(...) to simulate a quick bounce upward upon clicking the jump key (up or space) and also sets the alarm to 30 (with default speed, 1 second). The remaining code simply destroys and recreates the object if it goes outside of the bounds of the room.



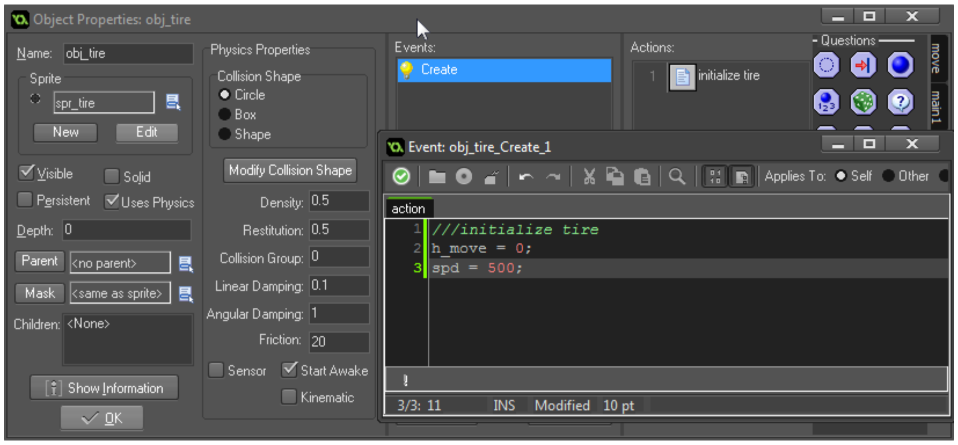
```
///hero step events

//control movement
h_move = keyboard_check(vk_right) - keyboard_check(vk_left);

if h_move != 0 {
    tire1.h_move = h_move;
    tire2.h_move = h_move;
    dir = clamp(h_move*100, -1, 1);
    with(tire1){
        physics_apply_torque(spd*h_move);
    }
    with(tire2){
        physics_apply_torque(spd*h_move);
    }
    physics_apply_torque(spd*h_move);
}

//control jump
jump_key = keyboard_check_pressed(vk_up) || keyboard_check_pressed(vk_space);
if (alarm[0] <= 0 && jump_key) {
    physics_apply_local_impulse(0, 0, 0, -200);
    alarm[0] = 30;
}

//death event
if (y >= 960 || x <= -60 || x >= 1660) {
    instance_create(500, 0, obj_hero);
    with(tire1){
        instance_destroy();
    }
    with(tire2){
        instance_destroy();
    }
    instance_destroy();
}
```


Object Properties: obj_tire

Name: obj_tire
Sprite: spr_tire
Visible ✓ Solid
Persistent Uses Physics ✓
Depth: 0
Parent: <no parent>
Mask: <same as sprite>
Children: <None>

Physics Properties
Collision Shape: Circle
Density: 0.5
Restitution: 0.5
Collision Group: 0
Linear Damping: 0.1
Angular Damping: 1
Friction: 20
Sensor Start Awake ✓ Kinematic

Events: Create
Actions: 1 initialize tire

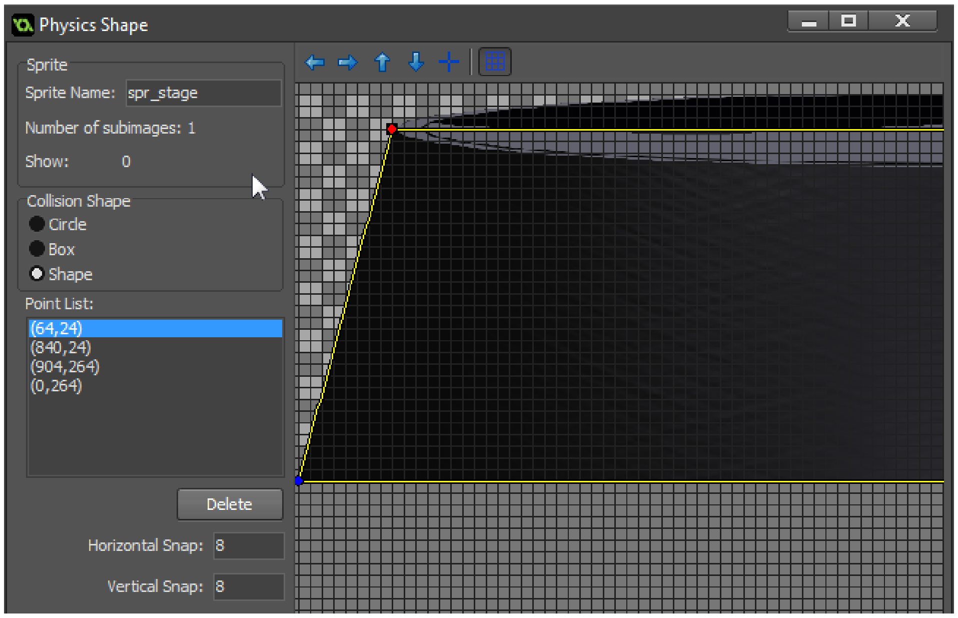
Event: obj_tire_Create_1

```
///initialize tire
h_move = 0;
spd = 500;
```

3/3: 11 INS Modified 10 pt

For the tire objects, we will only need a “Create” event that will run code initializing 2 variables. Similar to the hero object, we need “h_move” and “spd”.

Just as we modified the collision shape of the hero before, we also need to change the collision shape of the stage to fit how we want it to behave with the hero.
Now that we have the objects for the hero and the stage, we need to create an object that we will set as the parent for both.

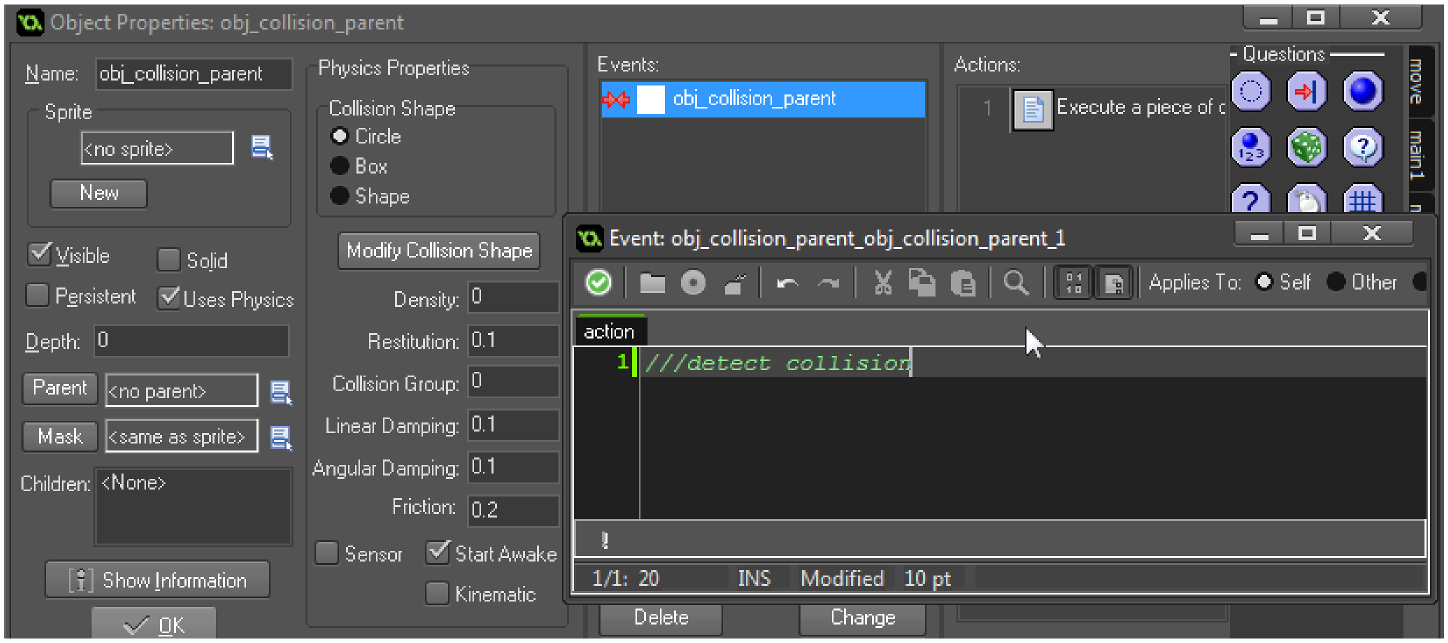

Physics Shape

Sprite
Sprite Name: spr_stage
Number of subimages: 1
Show: 0

Collision Shape: Circle Box [✓ Shape]

Point List:
(64,24)
(840,24)
(904,264)
(0,264)

Horizontal Snap: 8
Vertical Snap: 8


Object Properties: obj_collision_parent

Name: obj_collision_parent
Sprite: <no sprite>
Visible ✓ Solid
Persistent Uses Physics ✓
Depth: 0
Parent: <no parent>
Mask: <same as sprite>
Children: <None>

Physics Properties
Collision Shape: Circle
Density: 0
Restitution: 0.1
Collision Group: 0
Linear Damping: 0.1
Angular Damping: 0.1
Friction: 0.2
Sensor Start Awake ✓ Kinematic

Events: obj_collision_parent
Actions: 1 Execute a piece of c

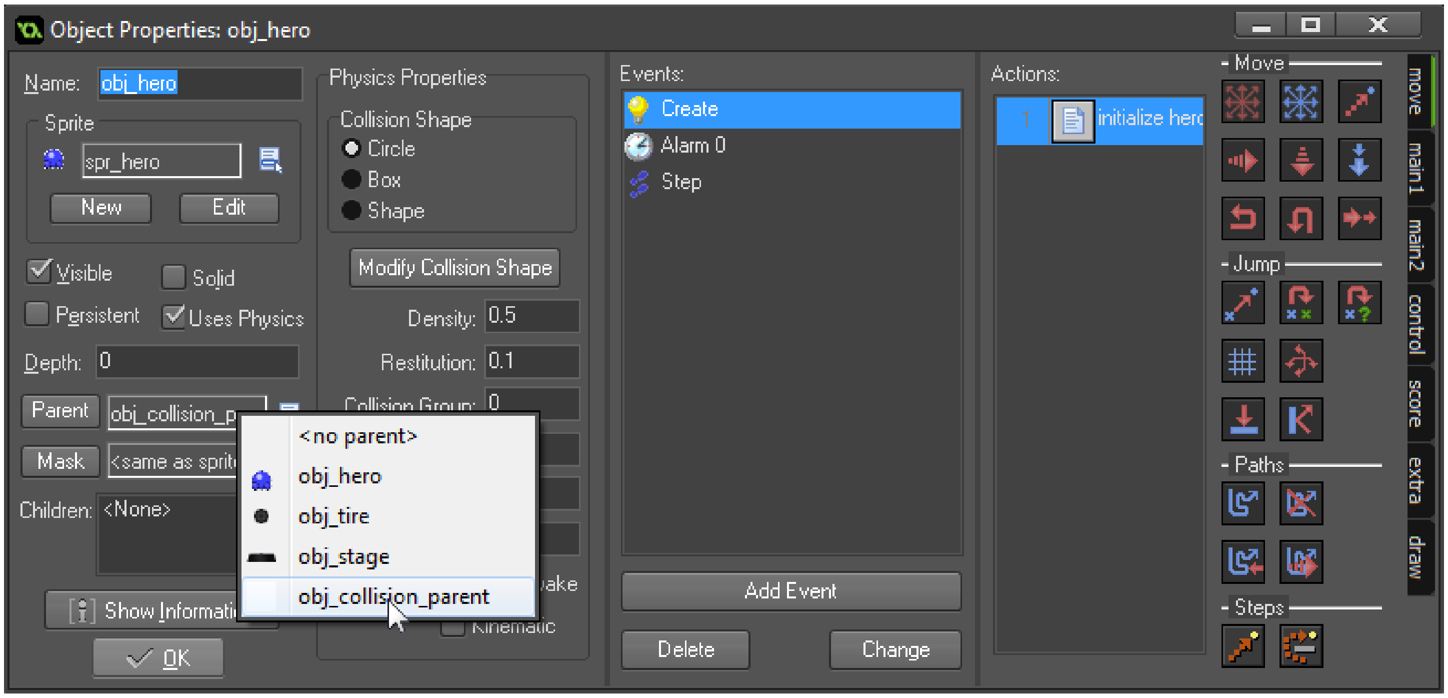
Event: obj_collision_parent_obj_collision_parent_1

```
///detect collision
```

1/1: 20 INS Modified 10 pt

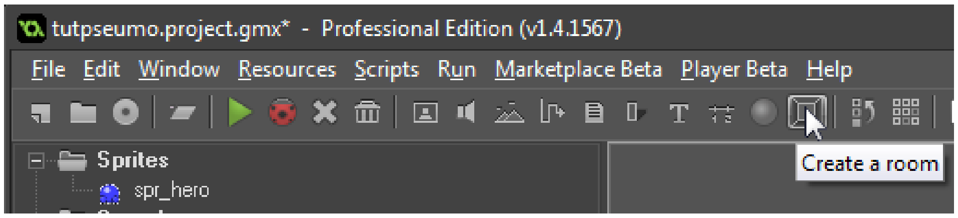
The collision object will allow for the hero and the stage to collide with one another, giving the effect of the hero falling onto the stage and being able to move around upon it. Click the checkbox for “Uses Physics” and add an event that will run on collision with itself. Set the action for this event as code that contains the comment “///detect collision”. This will allow all children of the collision parent to collide with one another.

In order for the hero and the stage to properly collide with each other, we will need to set each of their parents to the collision object we just created.

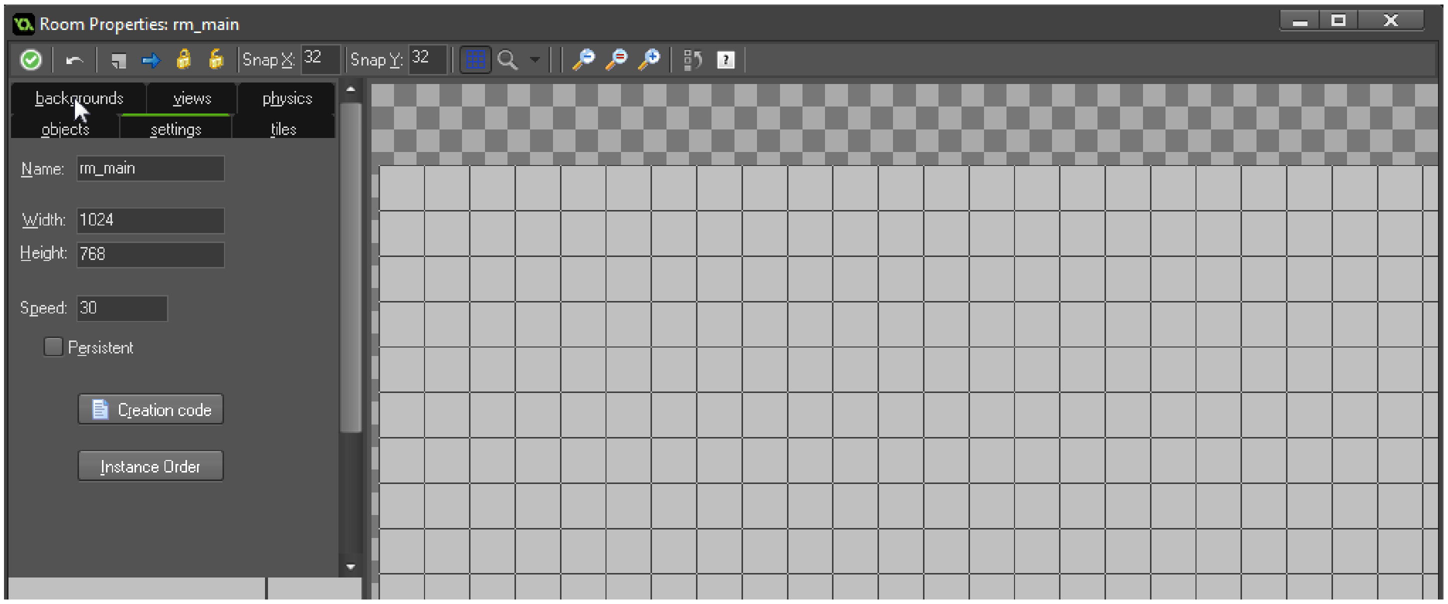


Creating the Main Room and Placing the Objects

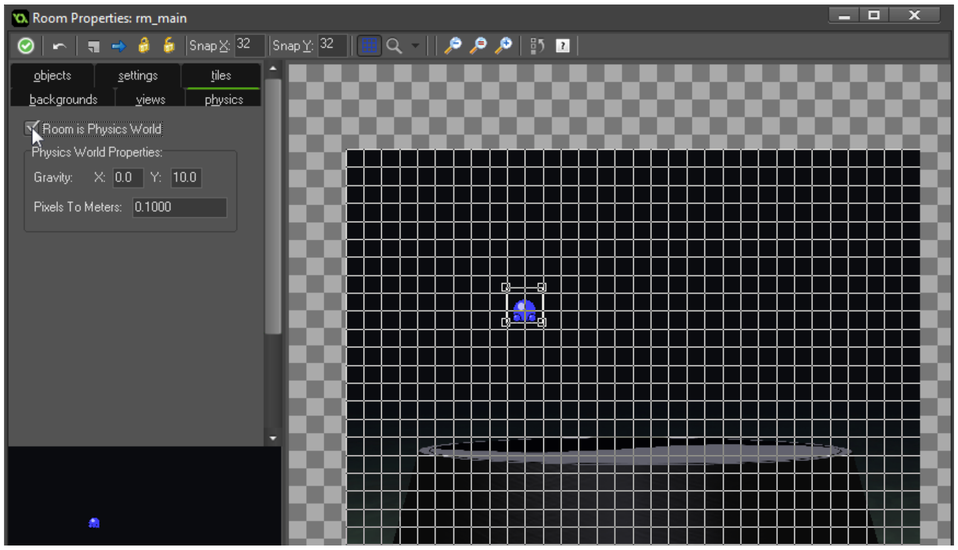
Now that we have our objects completed, we will need to create a room where we can place them. The "Create a room" button is next to the "Create an Object" button at the top bar. This will open a new window that controls the settings for a new room.


tutpseumo.project.gmx* - Professional Edition (v1.4.1567)
File Edit Window Resources Scripts Run Marketplace Beta Player Beta Help
Sprites
spr_hero
Create a room

Creating the first room will make it the default room that opens upon starting the game.



After creating the room, we will first need to set the background the one we created before. This is achieved by clicking the “backgrounds” tab and then selecting our background. We must also check the box that enables the room the be a “Physics World” so that the objects with physics enabled will behave properly.



Room Properties: rm_main

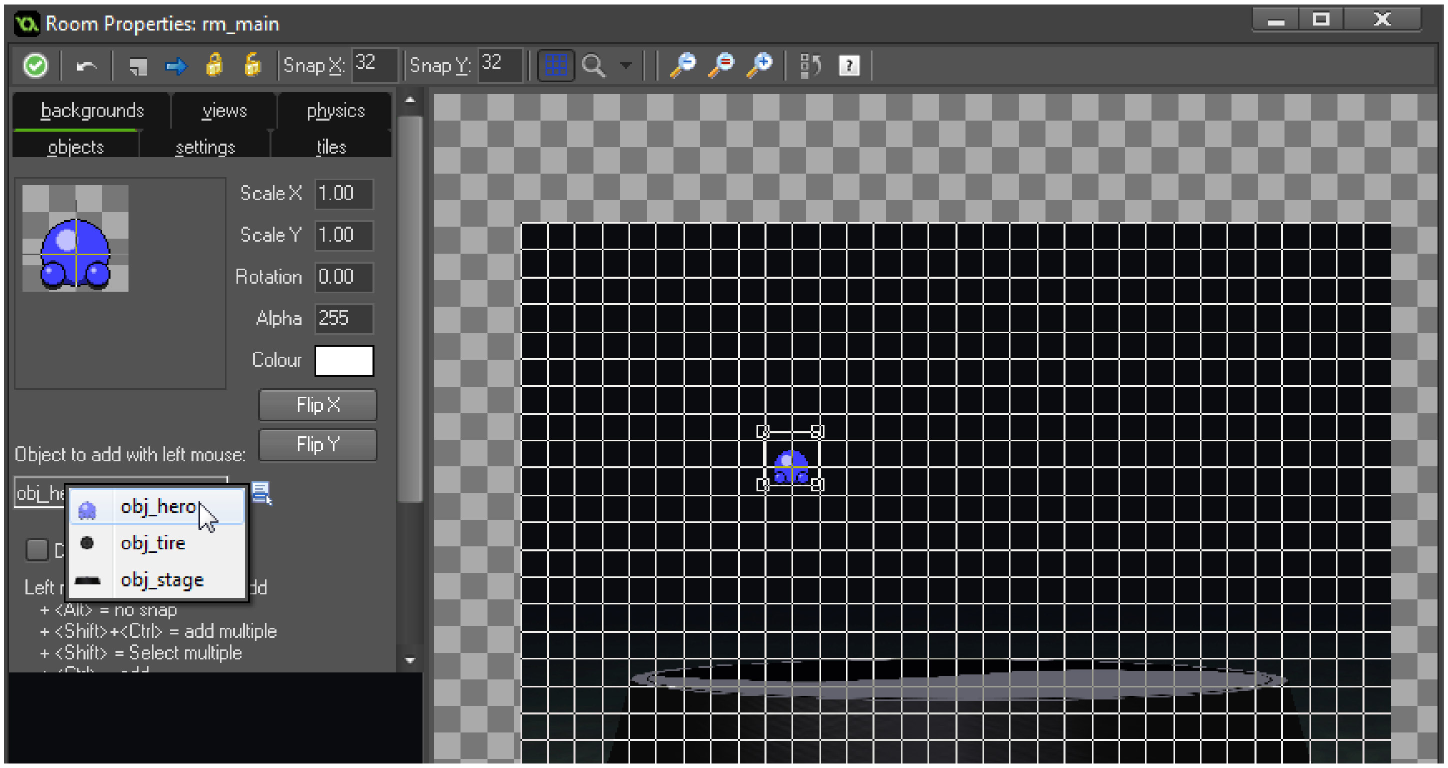
Snap X: 32 Snap Y: 32

Room is Physics World

Physics World Properties:

Gravity: X: 0.0 Y: 10.0

Pixels To Meters: 0.1000



Room Properties: rm_main

Snap X: 32 Snap Y: 32

Scale X 1.00

Scale Y 1.00

Rotation 0.00

Alpha 255

Colour

Object to add with left mouse:

obj_hero

obj_tire

obj_stage

+ <Alt> = no snap

+ <Shift>+<Ctrl> = add multiple

+ <Shift> = Select multiple

The final step is add the 2 objects to the room. Since we didn’t explicitly set the depth of each object, the order in which we add the objects is important. If we add the hero first, it will properly lay over the stage while still landing on the collision shape.

Conclusion

We now have a working room that holds a stage and a player-controlled object. This is a very simple example of what can be accomplished with GMS’s built-in physics engine. This may not be enough on its own to constitute a game, but its a solid start on a physics-based mechanic. It could be developed as a sumo-style king of the hill type game or a ball could be added for a traditional goal-based game.

GMS is a robust environment with plenty of options. This tutorial barely scratches the surface of its features. Fortunately, it’s used so commonly that there’s a deep well of resources, perhaps most notable youtube tutorials that cover nearly any 2D game concept that exists. If this hasn’t made the case for GMS’s viability, then it should be known that many of the most popular recent games have been built in GMS (Undertale comes to mind).