Ocerus User Guide

http://ocerus.sourceforge.net

February 17, 2011

Contents

1 Introduction

Welcome to the Ocerus user guide. With Ocerus you can create modern 2D games in an easy and straightforward manner. Ocerus can also handle 3D objects, so you can easily place your 3D models onto the 2D game plane.

The aim of this guide is to teach you the basics of the Ocerus usage, and by a simple project demonstration, it should help you set up and begin creating wonderful games in no time.

Throughout the user guide, we are going to show you basic game editing techniques on a sample pinball game, that you create from scratch. This guide concentrates only on real basics, so the game is rather simple but complex enough to demonstrate all necessary use cases.

2 Getting Ocerus

The Ocerus project is currently hosted at SourceForge and its homepage is `http://ocerus.sourceforge.net`. On that page, you can find project resources, such as the source code repository and installation packages. To download the latest installation package, navigate your browser to the Ocerus download page and download the latest version.

3 Installation

To install Ocerus, launch the installation file. Installation process is rather straightforward, so a few next button clicks will do the job. We are ready for our first launch.

4 First steps

To start Ocerus, click the Ocerus icon on your desktop or in the start menu. In a few moments, the main Ocerus window appears (see figure 1). Near the upper edge of the window there is the main menu. You can use this menu to carry out various actions in the editor. The rest of the window contains views that will be discussed later. If something is unclear you can see the description of all editor controls in the appendix C. For now let's start with creating a new project.

5 Creating the project

By selecting the **File** → **Create Project** command from the main menu, open the **Create Project** dialog. Type `Pinball` into the name field and browse to the location where your project will reside. Note that a new subdirectory in the chosen location will be created and it will be named the same as the project. That is why you should avoid using any nonstandard characters in the project name. When the **OK** button is clicked, the new project is created.

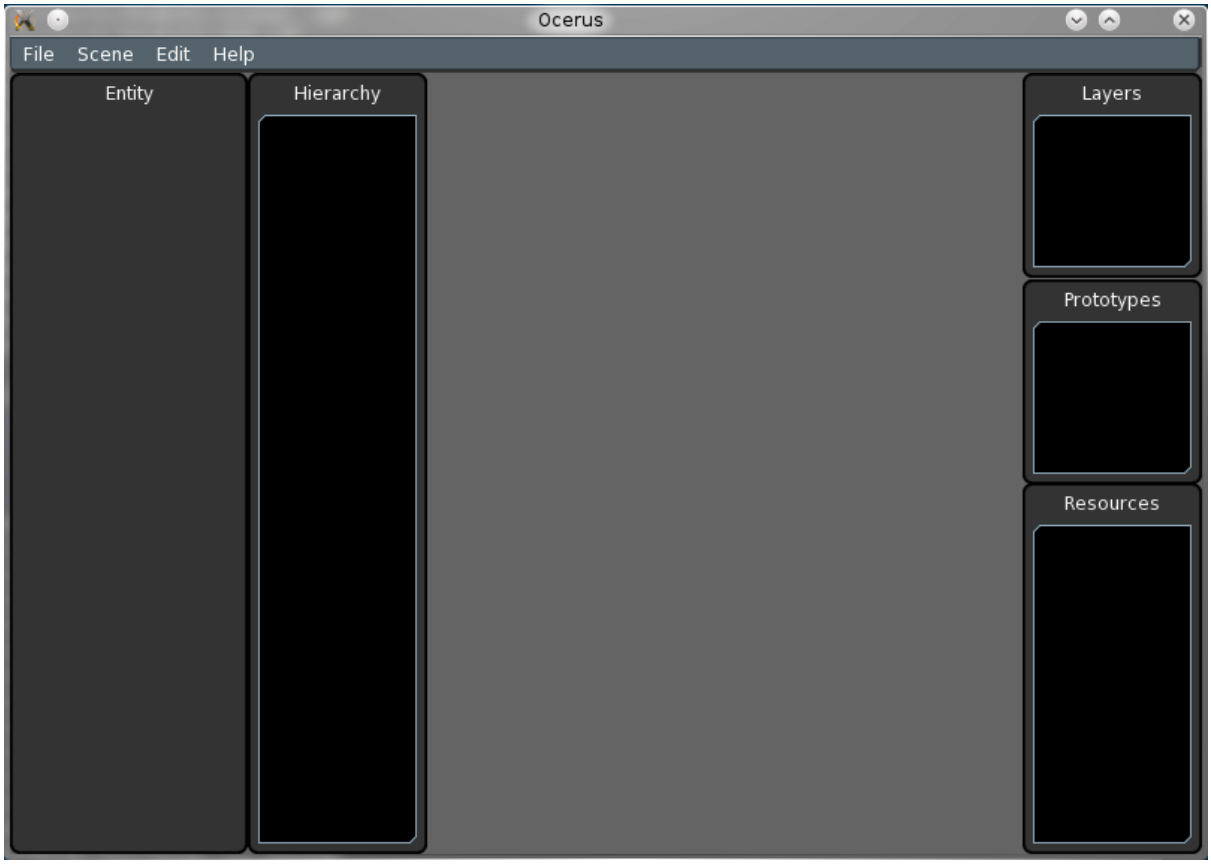

Figure 1: Ocerus window

A project in Ocerus represents a single game, so for every single game that you design, you need to create another project. Behind the scene, projects are ordinary directories in your file system with resource files representing your game content, such as images, models, scripts, etc. The structure of the project directory is on your own consideration, but we recommend to stick with the default structure that is created for new projects.

Although a default scene was automatically created together with the project, we are going to create another one. So let's create a **scene**.

6 Creating scenes

Select the **Scene** → **New Scene** command from the main menu. Ocerus can ask you whether you want to save the current scene (this is the default scene that was automatically created). In that case, click **No** and the **New Scene** dialog should appear.

In the **New Scene** dialog click on the folder icon to create a subfolder. Name it `scenes` and then enter this directory after it's created. Then type in `gameplay` as the scene filename (see figure 2). When you are ready, click the **OK** button and the new scene will be created and automatically opened.

Scenes are your virtual playgrounds where the game takes place. Generally speaking they consist of an indefinite plane with your game elements, such as the player's avatar

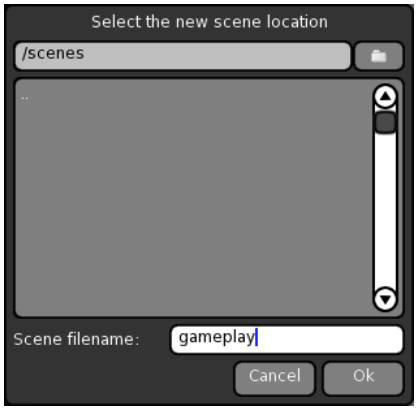

Figure 2: Creating a scene

or enemies, on it. You can think of a scene as a game level.

After a scene is opened, viewports and views are displayed (see figure 3). Viewports are framed areas that provide a view of the current scene. There are two viewports in the Ocerus editor. The **Game Viewport** (the upper one) shows the scene in the same way as it is rendered in the resulting game, whereas the **Editor Viewport** (the lower one) shows the scene with editing tools visible. Viewports are surrounded by views that provide information related to the current scene or the current entity. There are five views: **Entity**, **Hierarchy**, **Layers**, **Prototypes** and **Resources**. All of these will be discussed later. Now let's move on to create the protagonist of our game. You are right, we are talking about the ball.

7 Creating entities

To create an entity, select the **Edit** → **New Entity** command from the main menu and type the desired name of the entity into the edit box that has appeared. In our sample project there will be a ball that will bounce over the area, so we name the entity `Ball`. Click the **OK** button and the new entity appears in the **Hierarchy** view. Select the entity by clicking it in the view.

As soon as an entity is selected, the **Entity** view shows information related to it (see figure 4). First section in the view shows basic properties of the entity, such as its ID and its name. Some of these properties can be modified by clicking the property value and editing it. Note that if the property value is grayed, then it cannot be edited directly, or it cannot be edited at all.

Entities in Ocerus are composed of components that determine their behavior and functionality and these components can have other properties. All components and their properties are shown just under the **General** section in the **Entity** view and these properties can be edited the same way as the name property mentioned above.

Almost any entity that you will create is composed at least of the **Transform** component. Roughly speaking, entities with this component are such entities that are placed to a specific position in our virtual 2D world. They can move around, change its size and rotate. These transformations are represented by the **position**, **scale** and **angle**

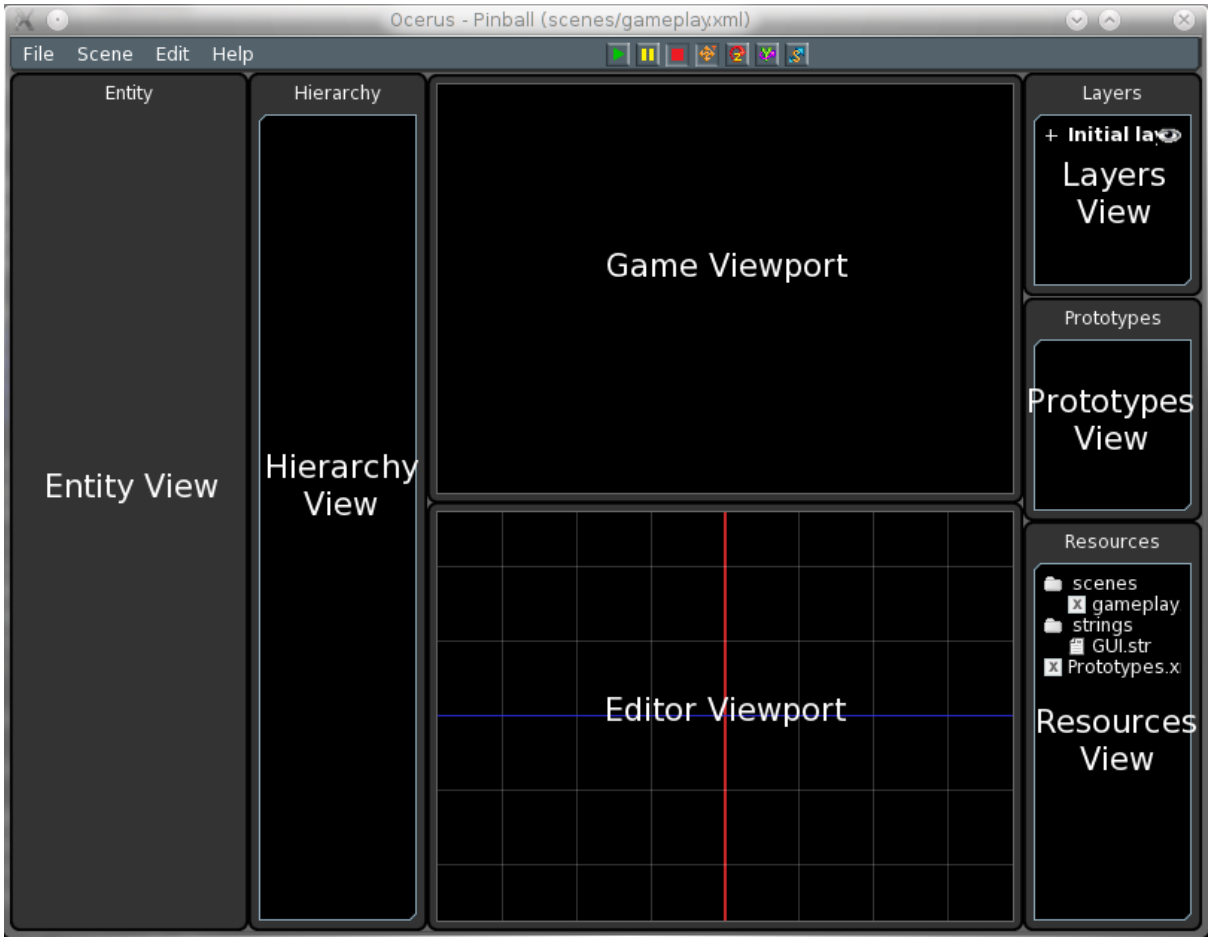

Figure 3: Ocerus with opened scene

properties. These properties are also reflected in the **Editor Viewport**. Although our ball does not have a graphics yet, it is represented in the viewport as a red square. You can try to modify the **position** property and see the red square move to another spot. However, if you change the **scale** property, you won't see any difference at all. We need to give our entity a visual form. This can be achieved with the **Sprite** component.

8 Creating components

Make sure that the `Ball` entity is selected, then click the **Edit** → **New Component** → **Sprite** action from the main menu. Our entity is now represented by a red filled square with the `NULL TEXTURE` text and a new section is appended to the **Entity** view (see figure 5). Notice that the red filled square is also visible in the **Game viewport**. This is because our entity has a visual form now. The default null texture is used, however. To give our ball a more appealing appearance, we have to change the **texture** property in its **Sprite** component. But how to get our fancy ball texture into Ocerus?

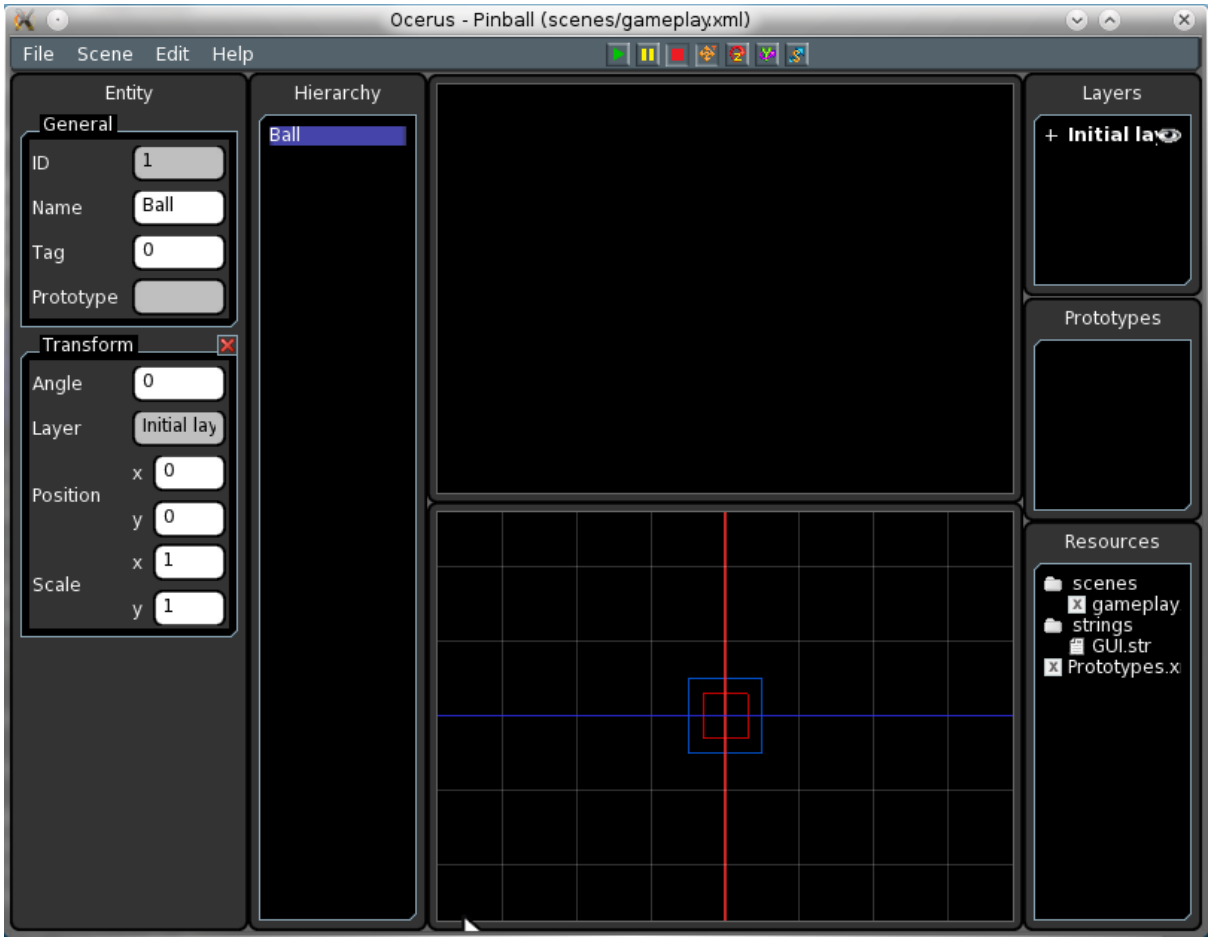

Figure 4: Newly created entity

9 Adding graphics content

Every file and directory that is placed into the project directory is visible in the **Resources** view. To put files with your textures into the project, simply copy them somewhere into the project directory. Remember that you cannot achieve this from Ocerus, so use your favorite file manager to do this.

When you add a file into the project directory, Ocerus automatically detects the file and creates an appropriate resource representing it. You can tell the type of the resource by the icon next to the resource name in the **Resource** view.

You can either use the `ball.png` texture from the `textures` directory in the archive provided with this document, or you can create or get any texture you like (see supported formats on page 24). Either way, copy it to the textures subdirectory of your project directory. Switch back to Ocerus and see that `ball.png` has appeared in the **Resources** view (see figure 6).

Now Ocerus is aware of our texture. Let's use it as a **texture** property of our entity.

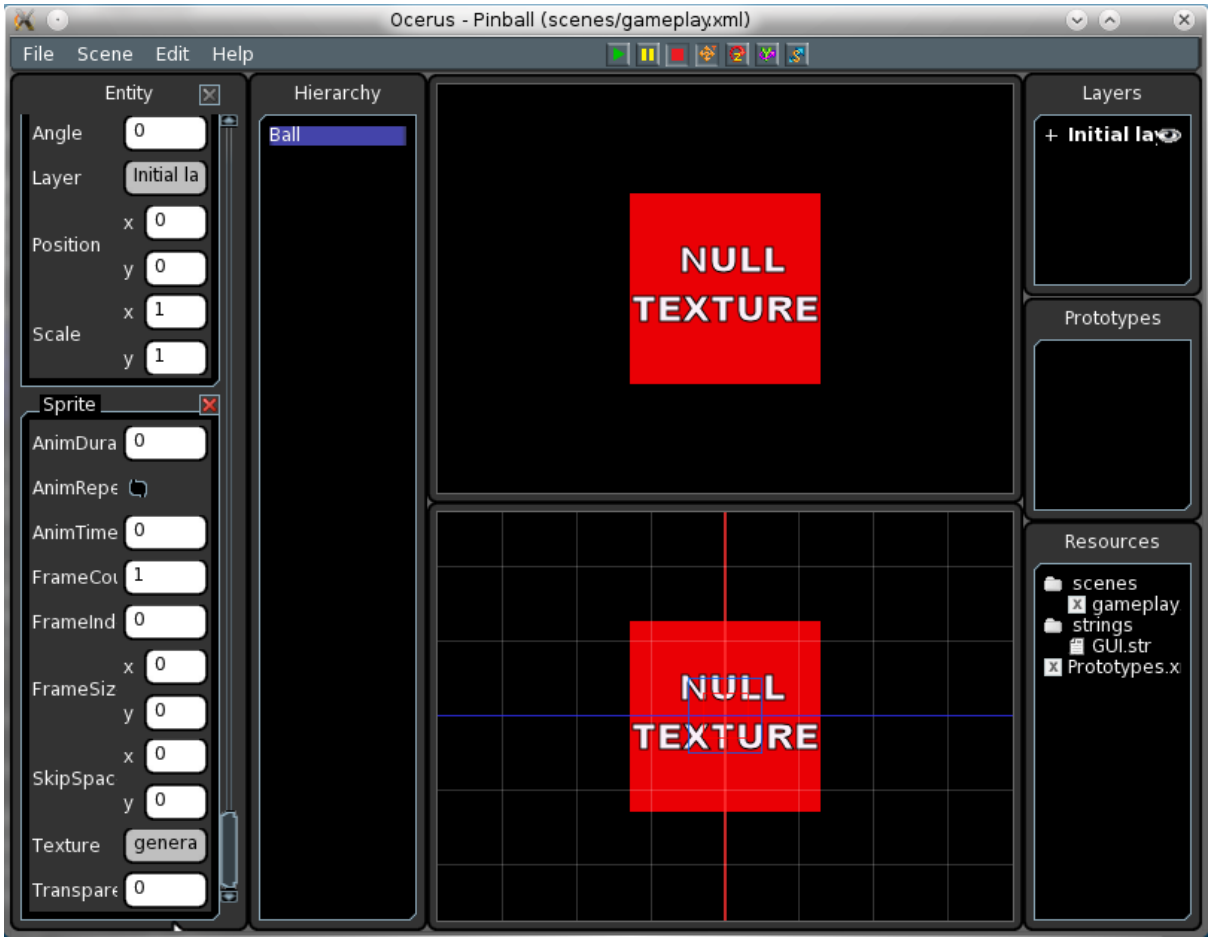

Figure 5: Entity with the Sprite component

10 Linking resources to properties

Properties that contain a resource (resource properties) cannot be edited with a simple text editor and therefore they are grayed. To link a resource to such property, you have to drag the resource and drop it to the grayed area of the property. Click the `ball.png` resource in the **Resource** view and drag it to **texture** property in the **Entity** view (see figure 6). If you were successful, you should end up with a ball image displayed in both viewports.

Now that we have an entity with a transform and a sprite component, we can take a look at how to use the editing tools.

11 Controlling the editor camera

To look around in your scene, you need to control the camera of the **Editor viewport**. Use the **arrow keys** on your keyboard to **move** the camera in the viewport. You can also move the camera with the **middle mouse button**. Simply click somewhere in the viewport with the middle mouse button and drag to move the camera. Finally you can

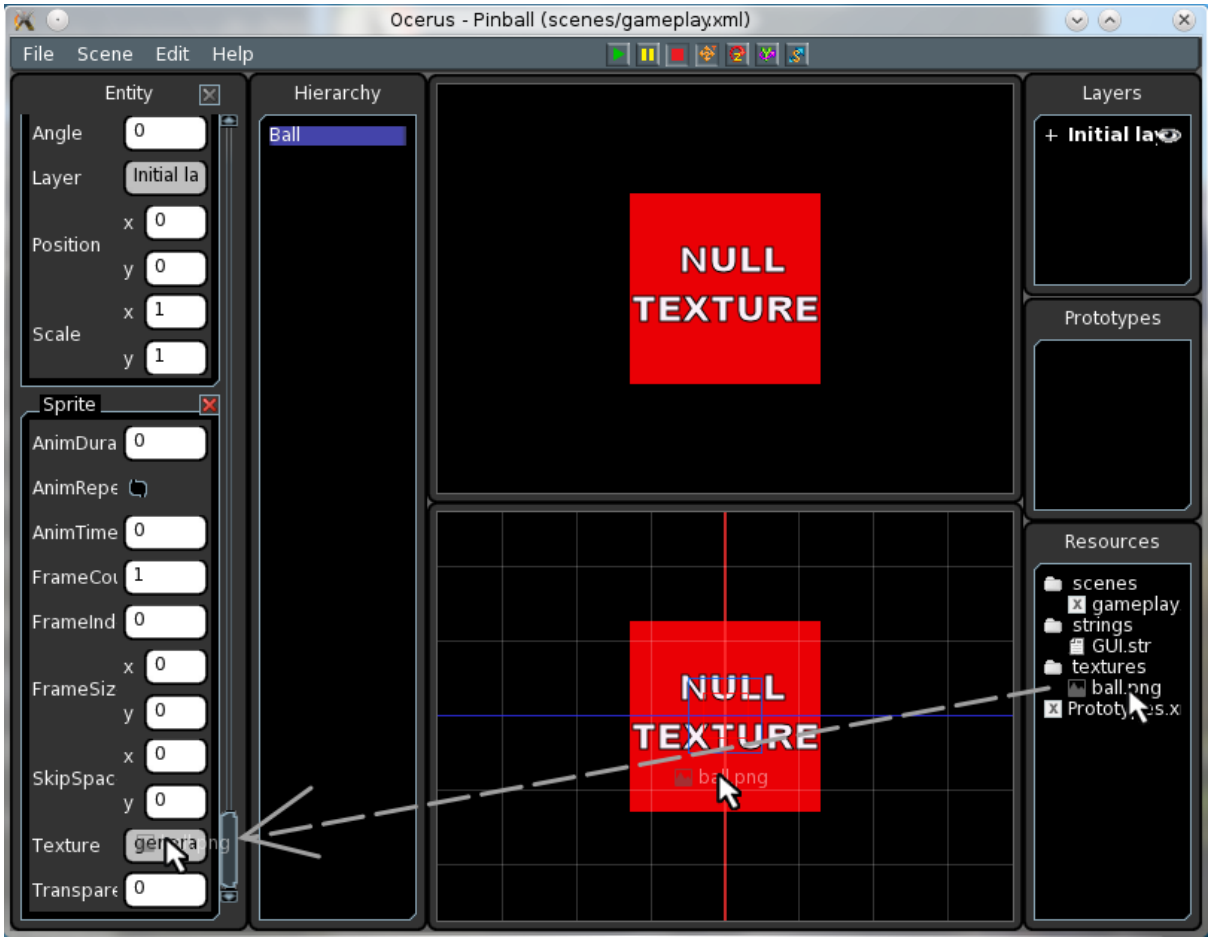

Figure 6: Linking texture to an entity

zoom the viewport using the **scroll wheel** on your mouse.

12 Moving, rotating and scaling entities

We have already discussed the way to move, rotate and scale an entity by editing its transform properties. This is, however, very uncomfortable. Ocerus provides editing tools to do this easily. There are four editing tools in Ocerus: the **move** tool, the **rotate** tool, the **rotate-y** tool and the **scale** tool. They are accessible from the toolbar located to the right from the main menu (see figure 7). You can switch between the editing tools by clicking the corresponding icon.

When the **move** tool is active, you can move the selected entity by dragging it in the editor viewport. By selecting the **rotate** tool and dragging the entity back and forth, you can rotate the entity and lastly the scale tool allows you to change the size of the entity, again, by dragging the entity in the editor viewport. Take some time to familiarize with these tools as they are vital for rapid development in the Ocerus. Consider using keyboard shortcuts to become even faster (click the **Help** → **Shortcuts** action in the main menu to display the list of keyboard shortcuts).

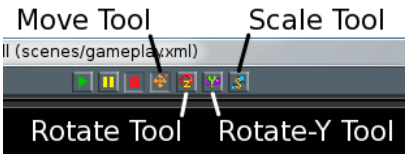

Figure 7: Editing Tools

13 Defining a collision polygon

Although our ball has a texture already, Ocerus has no clue what its shape is. As soon as we introduce physics into our game, it is necessary that every entity that is influenced by physics has a collision polygon defined. Ocerus needs to know these polygons in order to properly determine whether two entities collide with each other or not. To define a collision polygon, add the **PolygonCollider** component to our `Ball` entity (select **Edit** → **New Component** → **PolygonCollider**). The new component now appears under the **Sprite** component in the **Entity** view. The polygon is then defined as a list of points in the **polygon** property of the component. For our ball we will create an octagon with the following points: [0, -0,5], [0.35, -0.35], [0.5, 0], [0.35, 0.35], [0, 0.5], [-0,35, 0.35], [-0.5, 0] and [-0.35, -0.35].

Use the green plus button to add eight vertices to the **Polygon** property and then enter the points according to the figure 8. When you are ready, click the blue diskette button to save the list. After the new collision polygon is saved to our entity, it is displayed as a blue outline in the editor viewport. If you have used your own texture, you may need to modify these points to make the polygon aligned with the texture.

Our ball entity has a collision polygon now, which means that if physics is applied to that entity, it will collide with another entities accordingly to their collision polygons. So to really see the **PolygonCollider** in action, we have to add physics.

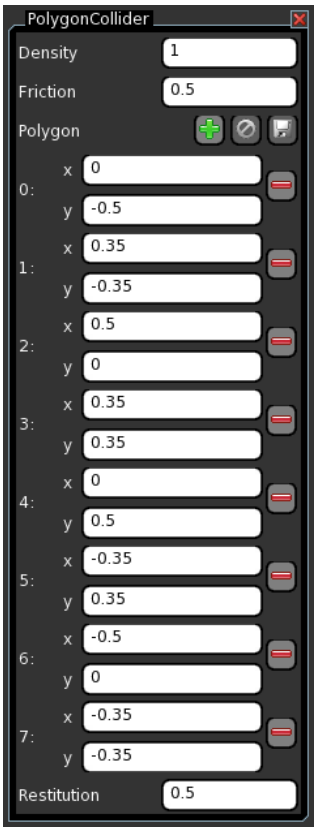

Figure 8: Defining octagon collision polygon

14 Adding physics

If you want an entity to be affected by the Ocerus physical engine, you need to add either of the following components to the entity: **StaticBody** or **DynamicBody**.

Entities with the **StaticBody** component are stationary building blocks in your world. They are never moved by the physical engine and their primary function is to build borders

and walls. On the other hand, entities with the **DynamicBody** component are used for objects that are affected by the gravitation force and other forces. Our ball entity is an example of an entity with the **DynamicBody** component.

By selecting the **Edit** → **New Component** → **DynamicBody** action, add the **DynamicBody** component to the `Ball` entity. Now the ball is affected by the gravitation force.

Ocerus does not implement the real gravitation force in the sense that all objects attract themselves. Instead, all objects are dragged straight down. It means that as soon as we start the game our ball will infinitely fall down. To stop it we need to create some barriers in our world.

Task 1. Create a new entity called `Platform`, add **Sprite** component and use the `textures\steel.jpg` file from the archive as the texture. Add the **PolygonCollider** component and set the polygon points to [5.12, -5.12], [5.12, 5.12], [-5.12, 5.12], [-5.12, -5.12]. Add the **StaticBody** component and finally use the edit tools to shape the square into a slightly sloped platform and place it under the ball (see figure 9). If you have any problems with this task, please read the corresponding sections above.

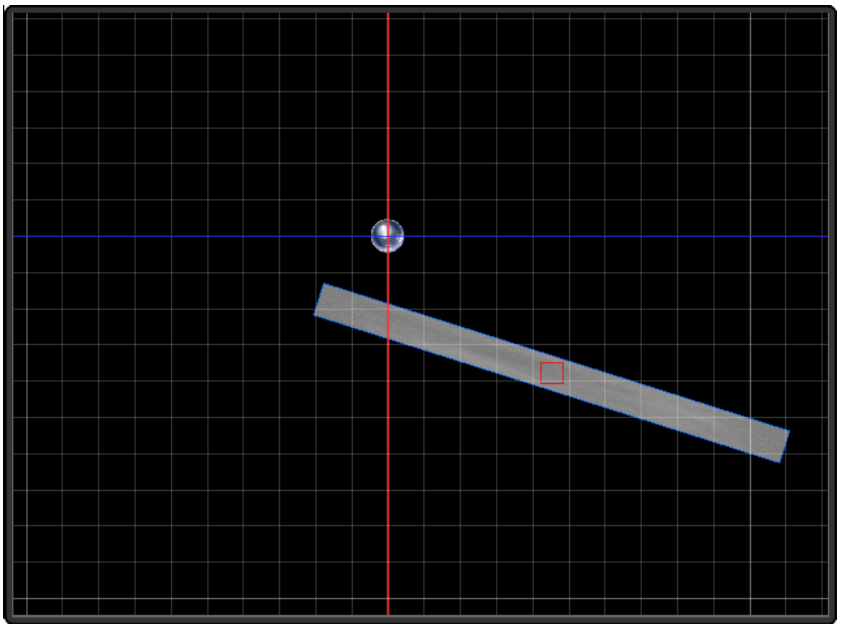

Figure 9: Task 1

When you are done, let's see what happens if we start the simulation.

15 Starting the action

While we were building our scene so far, everything was static even though we added physics to our entities. This is because the action is stopped. While editing you can arrange the initial state of your scene. Then, when you want to test what you have done, you can animate the scene by starting the action. Action is controlled by the action toolbar (see figure 10), which is composed of three buttons. Clicking the **Play** button

will start the action from the current state, clicking the **Pause** button will pause action in the current state and clicking the **Stop** button will stop the action and resets the scene to the initial state. After the action is started, you can still use the edit tools and edit entity properties. However, changes that you make will be discarded as soon as you stop the action.

Figure 10: Action Toolbar

Click the **Play** button and see the ball fall down on the platform. If your platform is sloped enough, you will see the ball roll down on the platform. You will probably see that the movement is not really smooth. This is because we used a collision polygon with only eight vertices for our ball. Try to double or even triple the number of the vertices and the movement will be smoother. Note that Ocerus does not provide colliders for round shapes.

Now it's time to design our pinball machine. The pinball machine will be composed of many platforms, so we need to create a lot of new entities. As you could see when you created the first platform entity, the process is not exactly short and you probably don't want to imagine how long it would take to create all those new entities the same way. For purposes of reusing existing entities there are few tricks in Ocerus. The first one is to use the **Edit** → **Duplicate Entity** action in the main menu. This action will duplicate the selected entity. You can duplicate the first platform entity a couple of times to have enough entities to design the pinball machine. Nevertheless, duplicating entities is not really recommended in this case.

Imagine that you decide to change the texture property of all your platform entities one day. If you used the duplicate entity method, you need to modify the texture property of every platform entity in each scene in your project. This issue is addressed by using prototypes, that allow some properties to be shared among multiple entities.

16 Using prototypes

Prototypes are special entity templates. They are not related to a concrete scene, rather they are saved in the `Prototypes.xml` file and can be accessed from any scene in the project. When you need to create a specific entity many times, you can create it once and then create a prototype from it. Then you can use this prototype as many times as you like.

To create a prototype from the `Platform` entity, make sure it is selected and then select the **Edit** → **Create Prototype** action from the main menu. A new item appears in the **Prototypes** view. All prototypes in your project are displayed there and instantiation of prototypes is as easy as dragging the prototype from the **Prototypes** view to the

Editor viewport. Drag your new prototype to the viewport couple of times to have enough entities to build the pinball machine.

When an entity is created from a prototype, this entity is linked to that prototype. This means that changes made to the prototype are propagated to all entities linked to it. However, this behavior is limited to only shared properties.

17 Shared properties

Click the `Platform` prototype in the **Prototypes** view and look at the **Entity** view (see figure 11). On the left of every property there is a check box that controls, whether the property is shared. If you change a property that has this check box checked, the change will be propagated to its linked entities. This concept is very handy if you need to globally change some properties across several scenes in your project. You only have to change the prototype and Ocerus will do the rest.

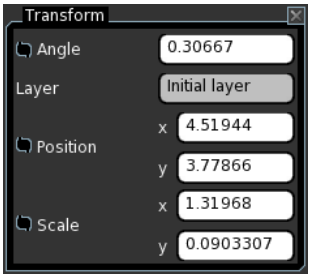

Figure 11: Shared Properties

Select the `Platform` entity from the **Hierarchy** view and take a look at the **Entity** view again (see figure 12). All properties that are shared have a lock icon to the right from them. That means they can only be edited in the prototype.

Figure 12: Locked Properties

Task 2. Now with the knowledge of prototypes and shared properties you should be able to design your pinball machine, so it would look similar to the figure 13. Use the `Platform` prototype to build all barriers. Also create the `Flipper` prototype that would be similar to the `Platform` prototype, except that it will use the `textures\yellow.jpg` texture from the archive and it will contain the **DynamicBody** component instead of the **StaticBody** component. Create two instances of this prototype called `LeftFlipper` and `RightFlipper`.

Hint: To create the ***`Flipper`*** *prototype you can instantiate the* ***`Platform`*** *prototype and unlink it by clicking the remove button on the right of the* ***Prototype*** *property in the* ***Entity*** *view. Then modify the entity and create a prototype from it.*

18 Entity hierarchy

As the number of entities in your scene grows, it becomes harder not to get lost. Fortunately, Ocerus provides means for hierarchical organization through the **Hierarchy** view. Unlike the file system, each and every entity can become a parent of other entities, so there are no folders in this concept. In fact, it's not an issue at all. Not only that you can use dummy entities (entities with no components) as folders, you are even encouraged to do so.

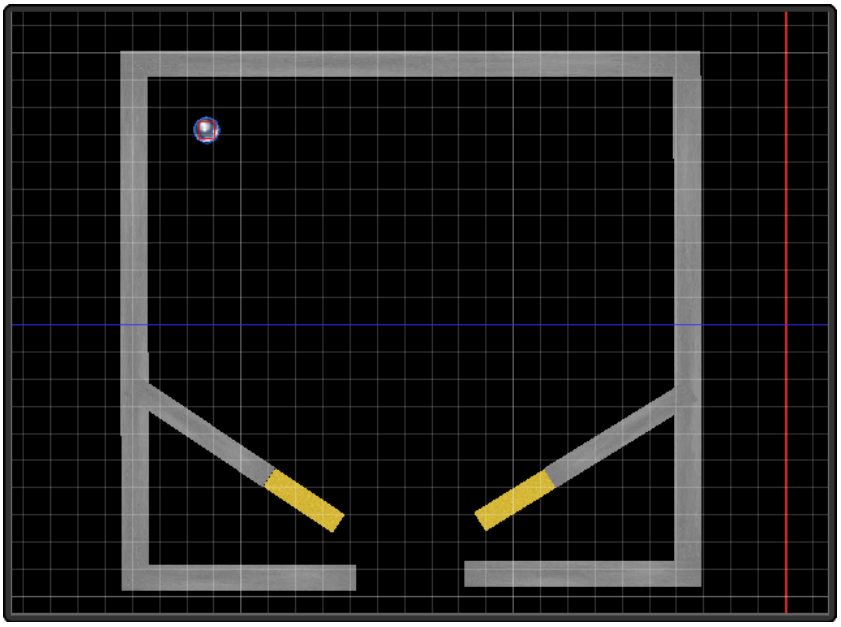

Figure 13: Task 2

Entities can be organized into a hierarchy by dragging them to their parents. This way you can move any entity to become a child of another entity. To move the entity to the top of the hierarchy (next to another top level elements), you need to use the **reparent up** action of the context menu couple of times. Just right click on the entity and select the **reparent up** action and the entity will move one level upwards in the hierarchy.

Entities can also be ordered among theirs siblings. Use the **move up** and **move down** commands to move an entity before its preceding sibling or after its succeeding sibling.

Now try to organize entities in your pinball scene. Create a dummy entity called `Platforms`, remove the **Transform** component by clicking the red cross to the right of the component title in the **Entity** view. Then move all platform entities to the Platforms entity. Do the same with flippers (see figure 14).

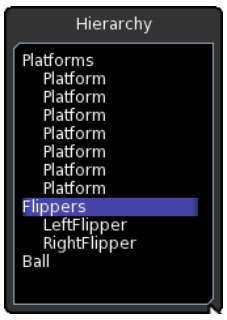

Figure 14: Hierarchy View

To further improve the visual appearance of the scene, we will add a background picture behind our pinball machine. Create an entity called `Background` and add the **Sprite** component to it. Use the `background.jpg` image as its texture. Notice that the `Background` entity is displayed above your pinball machine. So far we didn't have to take the order of sprites into account, but now we need to make sure that background stays behind the pinball machine. This can be achieved by layers.

19 Using layers

If you look at the **Layers** view, you can see the `Initial Layer` there. Every scene must have at least one layer and this one is automatically created. You can click on the plus sign to the left of the layer name to expand it. Every entity that you created so far is located in this layer. To make sure that our `Background` entity will stay behind all other entities, you have to create a new layer.

To create a layer, right click on an existing layer in the layer view and select the **new layer** command. A prompt asking for the layer name will pop up. Type `Background` and click the **OK** button. In case the `Background` layer is not under the `Initial Layer`, use the **move down** command in the layer's context menu. Finally expand both layers and drag the `Background` entity to the background layer (or use the **move down** command in the entity's context menu).

Notice that the initial layer is printed in boldface. This means that the initial layer the active one. If you are selecting an entity in the editor viewport, only entities in the active layer are selectable. To switch the active layer, double click on another layer and it becomes active. You can also toggle the layer visibility by clicking the eye icon to the right of the layer.

The visual appearance of our scene is done, as is physics. Now it's time to write the underlying scripts to implement the game logic.

20 Scripting

The Ocerus engine allows you to implement the game behavior using scripts. Although writing scripts is indeed programming, it differs from the core Ocerus programming in many ways. First, the Ocerus scripting environment was designed for game designers, rather than senior C++ programmers. It does not overwhelm you with unnecessary complexity of the "guts" of the engine, but it rather provides only the relevant things that are used in most cases. Also the scripting language is simpler than C++.

We are going to implement the main control of our game - the flippers. Make sure that you have already created the `Flipper` prototype, as well as both `Flipper` entities. Now select the `Flipper` prototype and add the **Script** component to it.

The **Script** component allows you to implement a custom behavior using the scripting language. As soon as you add the component, you can see the **ScriptModules** property in the **Entity** view. This property contains a list of script files that are associated with this entity. Now it's time to create our little script. Open your favorite text editor and copy the following code (or use the `scripts\flipper.as` file from the archive).

Listing 1: flipper.as

```
const float32 MAX_ANGLE_DELTA = 0.6f;
const float32 ANGLE_IMPULSE_RATIO = 10000.0f;
const float32 ANGLE_CHANGE_RATIO = 10.0f;



void OnPostInit()
{
  // register properties so that we can set them up in the editor... these are visible
```

```
  this.RegisterDynamicProperty_bool("LeftSided", PA_FULL_ACCESS, "True if the flipper is
      on the left side.");
  this.RegisterDynamicProperty_Vector2("Pivot", PA_FULL_ACCESS, "Location of the pivot
      point in local coords.");

  // these are internal variables
  this.RegisterDynamicProperty_float32("InitAngle", PA_SCRIPT_READ | PA_SCRIPT_WRITE, ""
      );
  this.RegisterDynamicProperty_Vector2("InitPivotPosition", PA_SCRIPT_READ |
      PA_SCRIPT_WRITE, "");
  this.RegisterDynamicProperty_bool("IsPressed", PA_SCRIPT_READ | PA_SCRIPT_WRITE |
      PA_EDIT_READ, "");
  // set initial values when the script is loaded
  Vector2 pivotWorldPos = this.Get_Vector2("Position") + MathUtils::RotateVector(this.
      Get_Vector2("Pivot"), this.Get_float32("Angle"));
  this.Set_Vector2("InitPivotPosition", pivotWorldPos);
  this.Set_float32("InitAngle", this.Get_float32("Angle"));
}

void OnUpdateLogic(float32 delta)
{
  // get current values
  float32 angle = this.Get_float32("Angle");
  float32 initAngle = this.Get_float32("InitAngle");


  // react on the keys and (indirectly) change the angle by applying an impulse to the
      object
  float32 impulse = ANGLE_IMPULSE_RATIO * delta;
  float32 change = ANGLE_CHANGE_RATIO * delta;
  if (this.Get_bool("LeftSided"))
  {
    if (gInputMgr.IsKeyDown(KC_LEFT) || this.Get_bool("IsPressed"))
    {
      // not entirely in the upper position yet
      if (angle > initAngle - MAX_ANGLE_DELTA) this.CallFunction("ApplyAngularImpulse",
          PropertyFunctionParameters() << -impulse);
    }
    else
    {
      // move the flipper down
      this.Set_float32("Angle", this.Get_float32("Angle") + change);
      this.CallFunction("ZeroAngularVelocity", PropertyFunctionParameters());
    }
  }
  else
  {

    if (gInputMgr.IsKeyDown(KC_RIGHT) || this.Get_bool("IsPressed"))
    {
      // not entirely in the upper position yet
      if (angle < initAngle + MAX_ANGLE_DELTA) this.CallFunction("ApplyAngularImpulse",
          PropertyFunctionParameters() << impulse);
    }
    else
    {
      // move the flipper down
      this.Set_float32("Angle", this.Get_float32("Angle") - change);
      this.CallFunction("ZeroAngularVelocity", PropertyFunctionParameters());
    }
  }
}

void OnUpdatePostPhysics(float32 delta)
{
  EnsurePaddleIsInBounds();
}

void EnsurePaddleIsInBounds()
{
```

```
  // eliminate eny unwanted velocities accumulating in our body
  this.CallFunction("ZeroLinearVelocity", PropertyFunctionParameters());

  // get current values
  float32 angle = this.Get_float32("Angle");
  float32 initAngle = this.Get_float32("InitAngle");

  // make sure the angle stays in the bounds
  if (this.Get_bool("LeftSided"))
  {
    if (angle > initAngle) angle = initAngle;
    if (angle < initAngle - MAX_ANGLE_DELTA) angle = initAngle - MAX_ANGLE_DELTA;
  }
  else
  {
    if (angle < initAngle) angle = initAngle;
    if (angle > initAngle + MAX_ANGLE_DELTA) angle = initAngle + MAX_ANGLE_DELTA;
  }

  // set new values to the object
  if (angle != this.Get_float32("Angle"))
  {
    this.Set_float32("Angle", angle);
    this.CallFunction("ZeroVelocity", PropertyFunctionParameters());
  }
  Vector2 pivotLocalPos = MathUtils::RotateVector(this.Get_Vector2("Pivot"), angle);
  Vector2 newPos = this.Get_Vector2("InitPivotPosition") - pivotLocalPos;
  this.Set_Vector2("Position", newPos);
}
```

Now save the file as `flipper.as` and copy it into the `scripts` subdirectory in your project. Back in Ocerus, select the `Flipper` prototype and click the **Add** button next to the **ScriptModules** property. Drag the `flipper.as` resource from the **Resources** view to the **NULL** field in the modules list and click the **Save** button. If everything went well, your script gets loaded and a couple of new properties appear in the **Script** component part of the **Entity** view. These properties are defined in the script and they allow you to control the scripted behavior. In our script, there are two properties: **LeftSided** and **Pivot**. The **LeftSided** property determines which flipper is the left one and which is the other one and the **Pivot** property defines the location of the pivot point.

Select the `LeftFlipper` entity and check its **LeftSided** property checkbox.

When you are ready, you can see your flippers in action. Start the action by pressing the **Play** button on the toolbar and try to control your flippers with the **Left** and **Right** keys.

To get more info on how to write scripts please see the `script reference` documentation.

20.1 Includes

Including other script files is a bit advanced concept which you don't necessarily need to understand but which comes handy from time to time. It allows you to split the code into manageble chunks and share functionality with other script files. You can include another script file into the current file by placing the following line to the top of the file:

```
#include "fileToInclude.as"
```

What the command does is it basically pastes the content of `fileToInclude.as` at the position of the command in the current script file. So when you use this command in

multiple script files you can share the code inside `fileToInclude.as`. Note that you must also include the subfolders in the path to the included file.

21 GUI

The last topic that is covered in this user guide is GUI creation. Ocerus uses the CEGUI library that provides the GUI functionality. It uses several files that define particular building blocks of your GUI. These are the **Looknfeel** files, the **Font** files, the **Imageset** files, the **Layout** files and the **Scheme** files. All these files can be edited with a simple text editor as they are just XML files, however CEGUI provides visual editors for some of them. You can download them at the CEGUI website.

The **Looknfeel** files define the visual appearance of the GUI elements. As a reasonable default **Looknfeel** is provided to you, we will not cover editing these files. You can find more on this topic in the CEGUI manual.

The **Font** files allow you to define fonts for your GUI, whether they are TrueType fonts or bitmap fonts. Again, Ocerus provide a reasonable default font so neither **Font** files are discussed in detail here. See the CEGUI manual to get more info about the **Font** files.

The **Imageset** files allow to define your custom images that can be used in GUI. CEGUI allows you to put multiple images into a single big image file and then specify which parts of the big image correspond to particular images. This map is defined in the **Imageset** files. In our `Pinball` project, we will define faces of two buttons that allow the player to trigger flippers by clicking them. Look at the **Imageset** file for our buttons.

Listing 2: buttons.imageset

```
<?xml version="1.0" ?>

<Imageset Name="Buttons" Imagefile="gui/buttons.png">
        <Image Name="LeftFlipperOn" XPos="0" YPos="0" Width="128" Height="128" />
        <Image Name="LeftFlipperOff" XPos="128" YPos="0" Width="128" Height="128" />
        <Image Name="LeftFlipperHover" XPos="256" YPos="0" Width="128" Height="128" />

        <Image Name="RightFlipperOn" XPos="0" YPos="129" Width="128" Height="128" />
        <Image Name="RightFlipperOff" XPos="128" YPos="129" Width="128" Height="128" />
        <Image Name="RightFlipperHover" XPos="256" YPos="129" Width="128" Height="128" /
            >
</Imageset>
```

The `Imageset` tag of the **Imageset** file specifies the name of the **Imageset** and the file that contains the images. It consists of several `Image` tags, that specify the particular images; their name, size and position within the image.

The **Layout** files define the layout of your GUI elements. Take a look at the **Layout** file for our scene that defines two flipper buttons. For now you can just use it as it is and we'll explain what it does later.

Listing 3: gameplay.layout

```
<?xml version="1.0"?>

<GUILayout>
<Window Type="DefaultWindow" Name="GameLayout">
        <Property Name="UnifiedAreaRect" Value="{{0,0},{0,0},{1,0},{1,0}}" />
        <Window Type="Pinball/ImageButton" Name="GameLayout/FlipperLeftButton">
                <Property Name="UnifiedAreaRect" Value="{{0,2},{1,-52},{0,52},{1,-2}}" /
                    >
```

```
                <Property Name="NormalImage" Value="set:Buttons image:LeftFlipperOn"/>
                <Property Name="HoverImage" Value="set:Buttons image:LeftFlipperOn"/>
                <Property Name="PushedImage" Value="set:Buttons image:LeftFlipperHover"/
                    >
                <Property Name="DisabledImage" Value="set:Buttons image:LeftFlipperOff"/
                    >
                <Event Name="MouseButtonDown" Function="OnFlipperLeftButtonDown"/>
                <Event Name="MouseButtonUp" Function="OnFlipperLeftButtonUp"/>
        </Window>
        <Window Type="Pinball/ImageButton" Name="GameLayout/FlipperRightButton">
                <Property Name="UnifiedAreaRect" Value="{{1,-52},{1,-52},{1,-2},{1,-2}}"
                    />
                <Property Name="NormalImage" Value="set:Buttons image:RightFlipperOn"/>
                <Property Name="HoverImage" Value="set:Buttons image:RightFlipperOn"/>
                <Property Name="PushedImage" Value="set:Buttons image:RightFlipperHover"
                    />
                <Property Name="DisabledImage" Value="set:Buttons image:RightFlipperOff"
                    />
                <Event Name="MouseButtonDown" Function="OnFlipperRightButtonDown"/>
                <Event Name="MouseButtonUp" Function="OnFlipperRightButtonUp"/>
        </Window>
</Window>
</GUILayout>
```

Finally, the purpose of the **Scheme** files is to group other data files and resources together, and to define some of their interactions. It usually contains a list of **Imageset** files, **Font** files and **Looknfeel** files.

However, so far we've only covered how the GUI looks but we still need to connect its events to gameplay. This is done using callbacks into a script. In our case we have the following script which reacts on button presses.

Listing 4: gameplay.as

```
void OnFlipperLeftButtonDown(Window@ wnd)
{
  EntityHandle flipper = gEntityMgr.FindFirstEntity("LeftFlipper");
  flipper.Set_bool("IsPressed", true);
}

void OnFlipperLeftButtonUp(Window@ wnd)
{
  EntityHandle flipper = gEntityMgr.FindFirstEntity("LeftFlipper");
  flipper.Set_bool("IsPressed", false);
}

void OnFlipperRightButtonDown(Window@ wnd)
{
  EntityHandle flipper = gEntityMgr.FindFirstEntity("RightFlipper");
  flipper.Set_bool("IsPressed", true);
}

void OnFlipperRightButtonUp(Window@ wnd)
{
  EntityHandle flipper = gEntityMgr.FindFirstEntity("RightFlipper");
  flipper.Set_bool("IsPressed", false);
}
```

You may have noticed that the GUI windows above contain the **Event** element. What it does exactly is described in the sections below but basically when an event is generated it attempts to execute the given function in the callback script attached to the layout.

To add the GUI to the project, copy the `gui` directory from the archive to the project directory. Then switch back to Ocerus and look at the **Resources** view. Ocerus tries to auto-detect types of the resources and usually does a great job. However, when it encounters the `.png` file, it identifies it as a **Texture** resource. To make it work with

CEGUI, the type of these image files must to be changed to the **CEGUI** resource. Right click the `buttons.png` file in the view and select the **Change Type** → **CEGUI** from the context menu. Now you are ready to put the layout with flipper buttons into the scene.

GUI layouts that are visible in your games are included into a scene by creating an entity with the **GUILayout** component. Create an entity named `GUILayout`, remove its **Transform** component and add the **GUILayout** component. Look at its properties (figure 15).

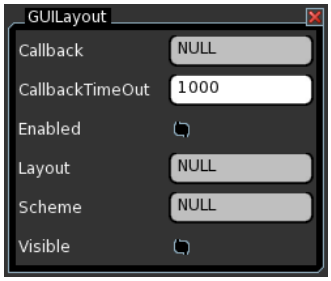

Figure 15: GUILayout component

First, drag the `pinball.scheme` file from the **Resource** view to the **Scheme** property in the **Entity** view. Then drag the `gameplay.layout` file to the **Layout** property. Now drag the `gameplay.as` script to the **Callback** property. Finally check the **Enabled** and **Visible** checkboxes. Now if you start the game the GUI should be working as promised!

21.1 Layout

Now it's time to understand what the layout file actually does and how it is structured. The GUI layout for the game should be defined in a file with the `.layout` extension and the XML internal structure. Since the CEGUI library is used for the GUI system a layout must fulfill its specification. It is broadly described in its documentation, so here's only a short description.

The `<GUILayout>` element is the root element in the XML file. It must contain a single `<Window>` element representing the root GUI element. The `<Window>` element must have the `Type` attribute which specifies the type of the window to be created. This may refer to a concrete window type, an alias, or a falagard mapped type. It can also have the `Name` attribute specifying a unique name of the window. This element may contain `<Property>` elements with the `Name` and `Value` attributes. They are used to set properties of the window. It may also contain `<Event>` elements with the `Name` and `Function` attributes which are then used to create bindings between the window and script functions (see section 21.3). Also, it can contain another `<Window>` elements as its child windows. For supported window types, properties and events see the CEGUI documentation.

21.2 String files

Note that there's a feature of the layouts specific to this engine. That is the translation of texts. While using the `Text` and `Tooltip` properties you can surround the value by $ (i.e. `Text=$text$`). That text is then used as a key to search for the text in the `GUI` group of the string manager. The string manager automatically loads the string files in the `strings` directory according to the currently selected language. More on that in the sections below.

21.3 Events

To be able to react when a user does something with the GUI you must connect the GUI layout to scripts. This can be done by placing the script into the **Callback** property of the **GUILayout** component. To subscribe to an event add the **Event** element to the window. Use its **Name** attribute to specify the event you want to react on. Use the **Function** attribute to set the name of the function in your script which will be executed when the event fires. The function must take a single parameter of the **Window@** type which holds a reference to the window which generated the event.

For example assume this is a part of a layout

```
<Window Type="CEGUI/PushButton" Name="Continue">
  ...
  <Event Name="MouseClick" Function="ContinueClick">
  ...
</Window>
```

and this is a part of a script module

```
void ContinueClick(Window@ window)
{
  Println(window.GetName());
}
```

connected via the **GUILayout** component together. The component can be in any active entity in the scene. Then every click on the button named **Continue** will print the message **Continue** into the log.

22 Strings

In the game you sometimes want to tell the player something. To keep the project simple it's better to separate the strings from the rest of the game. To do that the engine loads strings from the text files in the `strings/` directory. By default only the `GUI.str` file is there. If you open it you'll see it has a very simple structure. It consists of text items and comments. The pattern of a text item is **Key=Value**, where **Key** is an ID that is used for indexing a text representing **Value**. If **Key** equals **group** then it places all texts under that line into the group defined by **Value**. The **Key** must be unique within a group. Comments start with the # character. Here is an example of a correct text file:

```
# the name of the group to which the following texts belong
group=ExampleGroup
# comment to the text item below
example_key=Example text.
another_key=Another text.
```

The encoding of text files must be ASCII or UTF8. It is possible to end lines in the Windows (`\r\n`) or the UNIX (`\n`) style.

22.1 Directory layout

The directory in which the text files are stored must follow a specific layout. In the root directory there are files with default texts. These texts are used when the engine can't locate the text in the subdirectory of the current language. These subdirectories should contain the same files as the root directory. If any text varies in countries that use a common language it should be placed in a subdirectory of the language directory. It is recommended that the language and country names correspond with common norms. Here is an example of a directory layout:

```
-en
 -GB
  -textfile.str (1)
 -US
  -textfile.str (2)
 -textfile.str  (3)
-fr
 -textfile.str  (4)
-textfile.str   (5)
```

Assume that the files from the above example have the following content:

```
(1)
group=Group
country_specific=Country
(3)
group=Group
language_specific=Language
country_specific=Language
(5)
group=Group
default_string=Default
language_specific=Default
country_specific=Default
```

Now this table shows what will be actually displayed for different languages.

Text ID	Text value
Language: en-GB	
default_string	Default
language_specific	Language
country_specific	Country
Language: en	
default_string	Default
language_specific	Language
country_specific	Language
Language: default	
default_string	Default
language_specific	Default
country_specific	Default

23 Conclusion

Ok this is it! Now you have your first game fully working. What we've shown you is just a basic functionality of the editor but you can do much more, of course! The scripting system together with components is quite powerful to allow you to build different types of games. It might be also useful for you to take a look at the **Cube** project and explore what it actually does. Also, listed below are few sections about some general topic related to the editor itself and its behaviour from the user point of view.

A Resources archive

Resources needed to create the game described in this guide are accompanying the PDF. If you have opened the guide from the editor, the resources are likely to be in the `archive` directory. Otherwise, you need to download the archive package from the project website. See the Ocerus download page.

B Supported formats

Ocerus supports following file formats in your games.

Texture resources: JPEG, PNG, TGA, BMP, PSD, DDS, HDR
Model resources: OBJ[1]
CEGUI image resources: JPEG, PNG, TGA

[1] Content of the corresponding MTL file must be inserted to the beginning of the file. When exported from Blender, following options should be checked: "Copy Images", "Normals", "Polygroups", "Objects", "Groups", "Material Groups".

C Editor controls

In this part you can see the description of all features the editor can perform using menus and other mouse actions. First of all take a look at picture 16. There you can see parts of the editor which will be described in the sections below. What is not visible in the picture is a developer console. It can be opened and closed by pressing the ‘ key.

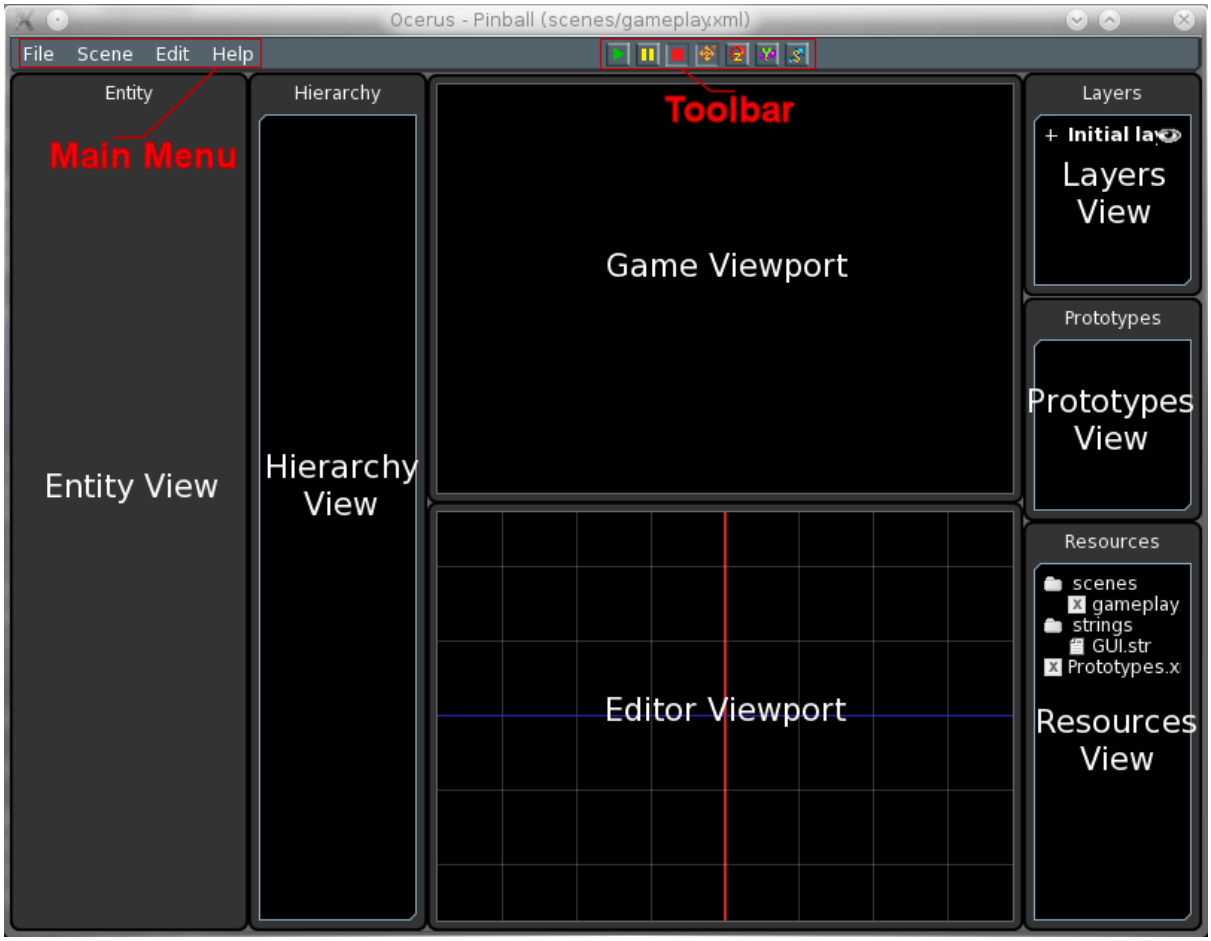

Figure 16: Different parts of the editor

C.1 Main menu

File	Function
Create Project	creates a new project with default resources
Open Project	opens an existing project from a folder location; the editor will need the project.ini file there
Close Project	closes the current project
Deploy Project	opens a dropdown menu with available platforms; after clicking on a platform and choosing a folder the application will be deployed there for the given platform
Quit	saves the project and closes the editor
Scene	Function
New Scene	creates an empty scene of the specified name
Open Scene	opens a list of available scenes; opens the scene after you click it
Save Scene	saves the current scene
Close Scene	closes the current scene
Edit	Function
New Entity	creates empty entity at the zero position
New Component	after you select the component type from the dropdown list it will be added to the current entity
Duplicate Entity	creates new entity identical to the currently selected entity at the place of the old one
Duplicate Selected Entities	if you have multiple entities selected this will duplicate them all
Delete Entity	deletes the currently selected entity
Delete Selected Entities	if you have multiple entities selected this will delete them all
Create Prototype	creates a prototype from the currently selected entity; the entity will be automatically linked to the prototype
Help	Function
About	info about the application
User Guide	this very document
Shortcuts	a list of shortcuts available from the editor
Script Reference	reference manual to the script system
Extending Ocerus	a document describing how to extend the funcionality of the engine
Design Documentation	a document describing the architecture of the engine and its parts
Doxygen	reference manual to the C++ engine classes

C.2 Toolbar

Action	these buttons control the game action
Play	starts or resumes the action; objects start moving
Pause	pauses the action if it's running
Restart	stops and rewinds the action; objects are reset to their state prior to the action start
Tools	these buttons control the current mouse edit tool
Move	mouse dragging moves selected objects
Rotate	mouse dragging rotates selected objects
Rotate Z	mouse dragging rotates selected objects along the Z axis; it's visible only if the objects have the Model component
Scale	mouse dragging scales the selected objects along the X and Y axes

C.3 Entity view

The entity view displays a list of components. Each component is represented by a box with a title of the type of the component. Inside of each of the boxes there's a list of their properties. You can influence the behaviour of components by changing these properties. You can do this from the editor but also from scripts.

There are different types of properties. Most of them can be edited as text, so you can just click them and type in the new value. And the boolean property can be changed by clicking on its checkbox. However, the resource property cannot be changed directly. To change it you must drag there a resource from the Resource view using your mouse and release it onto the property. The same way you take care of the prototype property (in the General component box) but instead of a resource you drag there an entity from the Prototype view. The layer property can only be changed by moving the object to a different layer in the Layer view.

Some of the properties cannot be changed at all. To find out read the property description. Hover your mouse over the property and wait until a tooltip with the description appears. Also, if the entity is linked to a prototype some of its properties might be marked as *shared*. It means that all prototype instances share the property and it can only be changed in the prototype itself. To know if a property is shared or not see if there's a lock icon next to it in the Entity view. You can mark/unmark properties shared if you select the prototype in the Prototype view. Then all of the properties has a checkbox next to them which does this thing.

When you save the scene all components and properties are automatically saved into the scene file. So when you reload the scene the properties get restored again. However, this doesn't work when the game action is running. After you restart the action all objects and properties get reset to the values before the action started.

C.4 Hierarchy view

Here you can see all entities in the game structured as a tree. If you click an item the entity gets selected and will appear both in the Entity view and in the Editor viewport as well.

If you double-click the item the Editor viewport will pan to the entity (if it has the Transform component, so to speak). You can drag the entities around in the Hierarchy view to change the structure of the tree. Right-clicking an item will bring up the following menu:

Move Up	moves the entity up in the list
Move Down	moves the entity down in the list
Reparent Up	moves the entity up in the hierarchy
Add Entity	creates an empty entity as a child of the current entity
New Component	adds a component to the entity
Duplicate Entity	duplicates the entity placing it at the same level in the hierarchy
Delete Entity	deletes the entity
Create Prototype	creates a prototype from the entity and links the entity to it

C.5 Game viewport

In the Game viewport the game is displayed as it will look in the result. Once the action is started (using the Toolbar) you can use your mouse and keyboard the same way you'll use them in the final game. Some editor controls and shortcuts are blocked until the action is paused or restarted.

C.6 Editor viewport

The Editor viewport allows you to select and edit entities and see the changes in the Game viewport. You can pan the camera using the arrow keys or by holding the middle mouse button and moving the mouse. Zoom out or in using the mouse wheel. Select entities by left-clicking them. You can select multiple entities by holding the shift key while left-dragging the mouse. Note that you can only select entities in the currently selected layer (in the Layer view). If you have an entity selected you can mouse-edit it by dragging it while the left button is down. The action what happens depends on what edit tool you have selected in the Toolbar. If you right-click and entity a context menu appears. It contains some of the items of the Edit submenu of the Main menu. They behave exactly the same, so read the description of the Main menu for more info.

C.7 Layers view

Here you see the list of layers in the game. Each entity must belong to a layer. Layers define what gets drawn first and also you can use them to filter out collisions. The topmost layer is drawn last. The current layer is displayed bold. You can change it by double-clicking another layer. If you click on the + button the layer gets unrolled, so you can see all entities in the layer. You can hide or show the layer by clicking the eyes icon. If you right-click a layer the following menu appears:

Move Up	moves the layer up in the list
Move Down	moves the layer down in the list
New Layer	adds new layer below the clicked one
Rename	renames the layer
Remove	deletes the layer

C.8 Prototypes view

This view lists all entity prototypes of the project. Note that the prototypes don't exist in any scene. They are global for the project. To instantiate a prototype you can use the context menu or you can just drag it to the Hierarchy view or the Editor viewport. You can also drag it to the prototype property of an entity in the Entity view. This will link the entity to this prototype. You can create prototypes either by using the prototype context menu (to create an empty one) or by using the context menu of an entity or the Edit submenu of the Main menu (to create the prototype based on the entity). Righ-clicking the Prototype view brings the following menu:

Create Entity	instantiates the prototype into a new entity; it appears at the bottom of the Hierarchy view
Add Prototype	creates an empty prototype
Delete Prototype	deletes the prototype and unliks all its instances

C.9 Resources view

Here you can see all resources usable in the project. It lists them in the same structure as they reside on the disk in the project folder. You can open a resource by double-clicking it. The default system editor will start. The exception is a scene resource which will open the scene in the editor. You can drag a resource to a resource property in the Entity view to change its value. Each of the resources has a specific type which limits its use in the project. The type is designated by an icon next to the resource name. The resource types are assigned automatically but you can force the change using the context menu:

Change Type	opens a list of available types for this resource; the current type has an asterix next to the name
Open Scene	opens the resource as a scene if it is a scene
Rename Scene	renames the resource as a scene if it is a scene

C.10 Developer console

The console can be opened and closed by pressing the ‘ key. It displays the editor log. It is usefull to see what's going on in the editor as well as scripts. If there is an error in the script it will be displayed here. Note that the editor log is automatically saved in the current folder in the *Core.log* file.

You can also use the console to try simple script commands. Simply type the command into the edit box and press enter. The command will be compiled and immediately executed.