# Ocerus: User Guide

Michal Cevora, Lukas Hermann,Ondrej Mocny, Tomas Svoboda

## Contents

**1 Introduction** **1**

**2 Installation** **1**

**3 First steps** **2**

**4 Creating the project** **2**

**5 Creating scenes** **3**

**6 Creating entities** **4**

**7 Creating components** **6**

**8 Adding graphics content** **7**

**9 Linking resources to properties** **7**

**10 Moving, rotating and scaling entities** **7**

**11 Defining a collision polygon** **8**

**12 Adding physics** **9**

**13 Starting the action** **11**

**14 Using prototypes** **12**

**15 Shared properties** **12**

**16 Entity Hierarchy** **14**

**17 Using layers** **14**

**18 Scripting** **15**

**19 GUI** **19**

# 1 Introduction

Welcome to the Ocerus User guide, a document that shows basic usage of the Ocerus Engine from the game designer perspective. With Ocerus you are able to 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 purpose of this document is to show you the basics of using Ocerus and by a simple project demonstration it should help you get set up and creating wonderful games in no time.

Throughout the user guide, the basic game editing techniques will be presented on a sample pinball game, that will be created from scratch. This guide concentrates only on real basics, so the game will be rather simple, but complex enough to demonstrate all necessary use cases.

# 2 Installation

The Ocerus project is currently hosted at SourceForge and its homepage is `http://ocerus.sourceforge.net`. On that page, you can find the brief description of the project, location of project resources, such as source code repository and installation packages. To download the latest installation package, navigate your browser to the Ocerus download page. Just download the installation package from there and launch it. Installation process is rather straightforward, so a few next button clicks will do the job. We are ready for our first launch.

# 3 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). In the top edge of the window there is 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. Let's start with creating a new project.

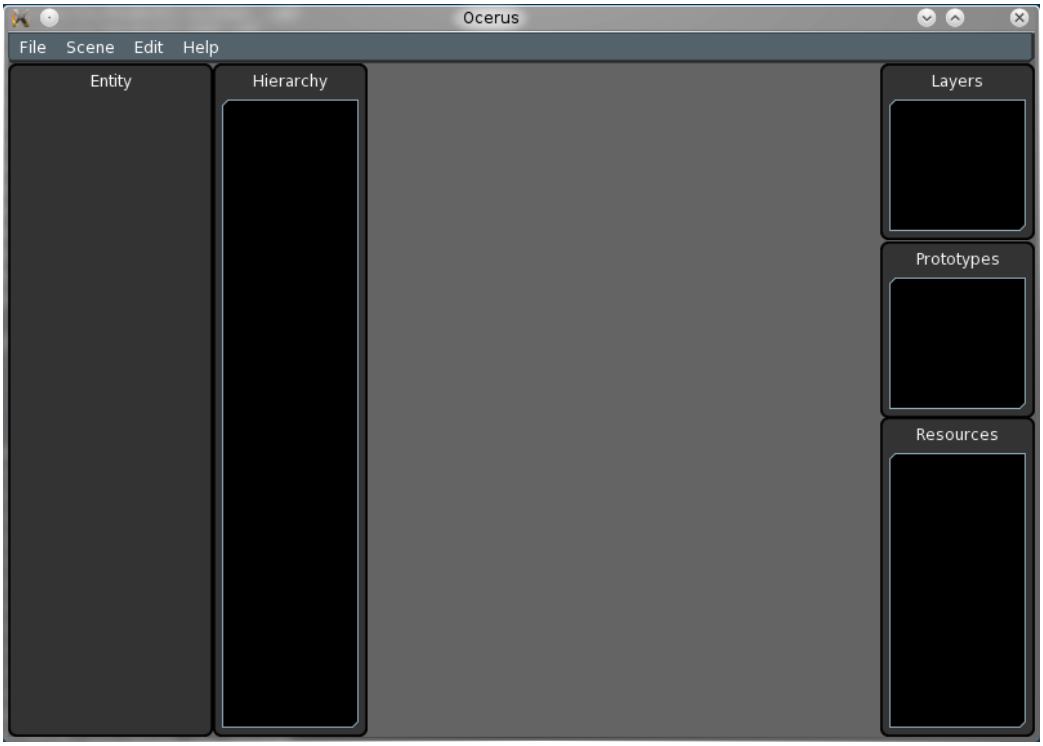


Figure 1: Ocerus window

# 4 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 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. Name the project `Pinball` and use any location you like. When the **OK** button is clicked, the new project is created.

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 on your file system with resource files representing your game content, such as images, models, scripts, etc. Structure of the project directory is on your own consideration, but it's recommended to stick with the default structure that is created for new projects.

The Ocerus window is still almost empty. Let's create a **scene**.

# 5 Creating scenes

Select the **Scene** → **New Scene** command from the main menu. In the **New Scene** dialog select the scenes sub-directory and type `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.

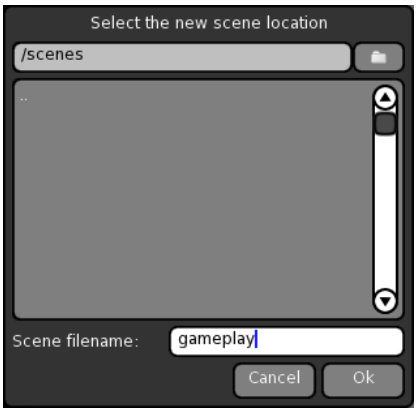


Figure 2: Creating a scene

After a scene is opened, viewports and views are displayed (see figure 3). Viewports are framed areas that provide a view to the current scene. There are two viewports in 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 with views that provide information related to current scene or current entity. There are five views: **Entity**, **Hierarchy**, **Layers**, **Prototypes** and **Resources**. All these views will be discussed later, now we move on creating an entity.

# 6 Creating entities

To create an entity, select the **Edit** → **New Entity** command from the main menu and type the entity name. 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 related information about it (see figure 4). First section in the view shows basic properties

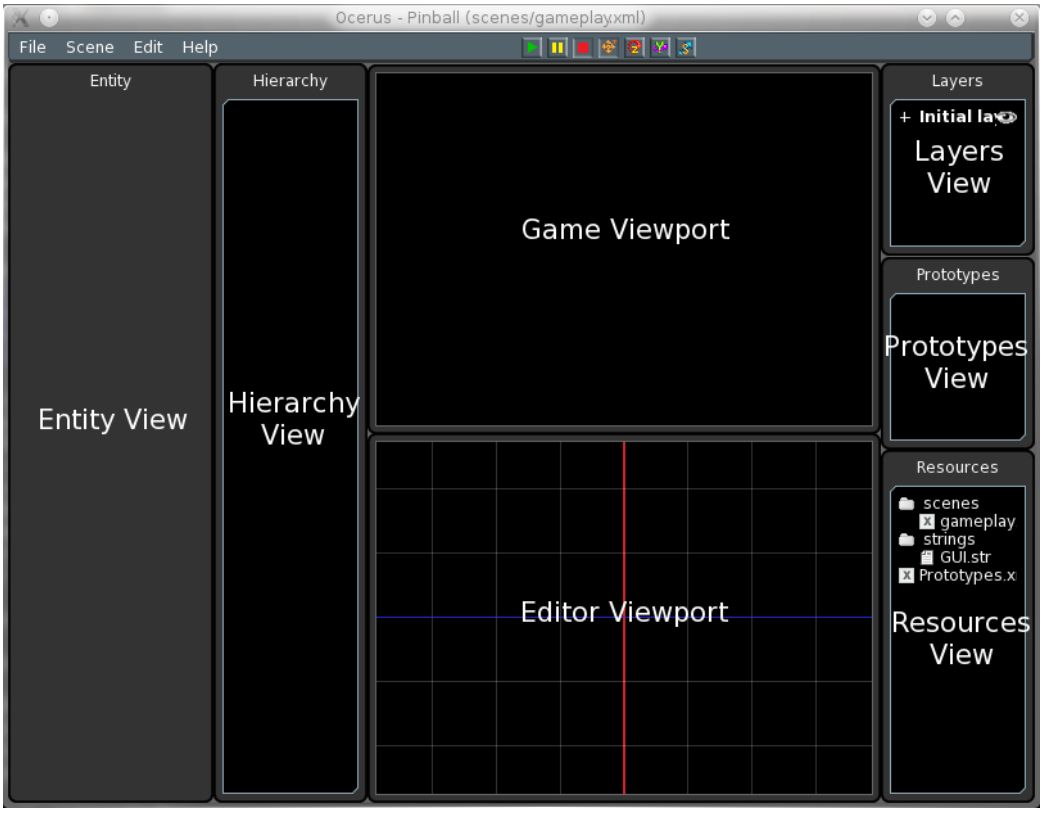


Figure 3: Ocerus with opened scene

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 another 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 ID and name property 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** 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

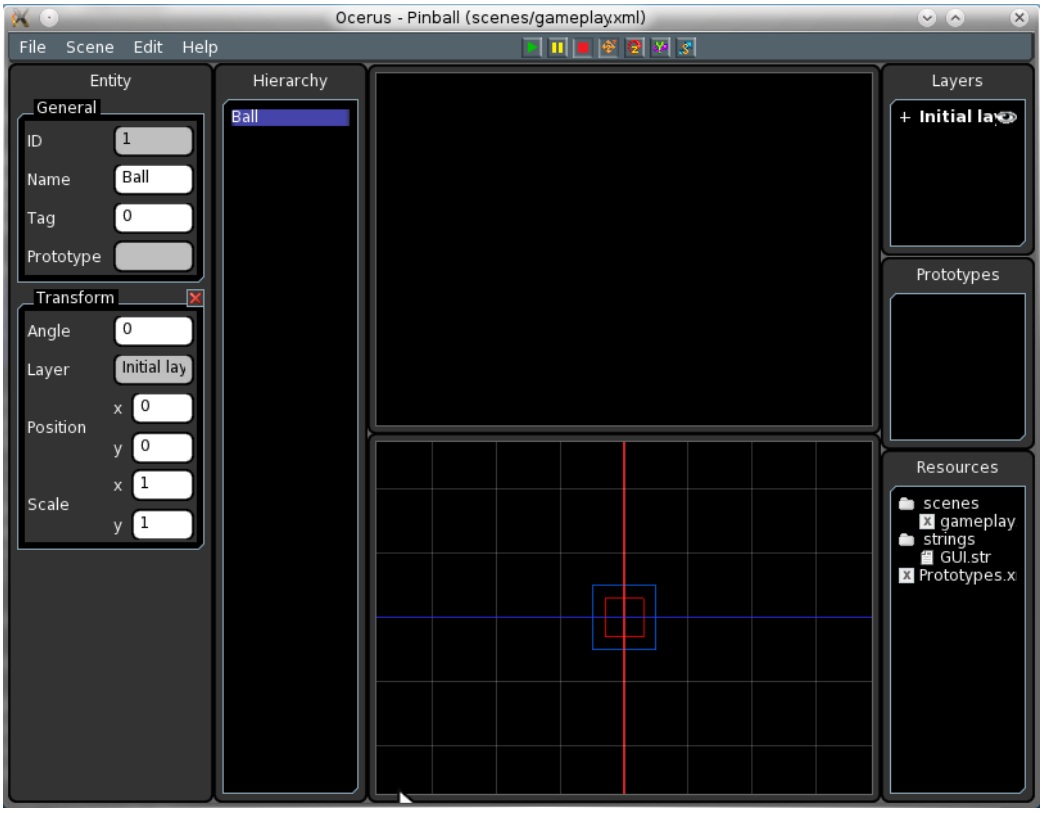


Figure 4: Newly created entity

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.

# 7 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?

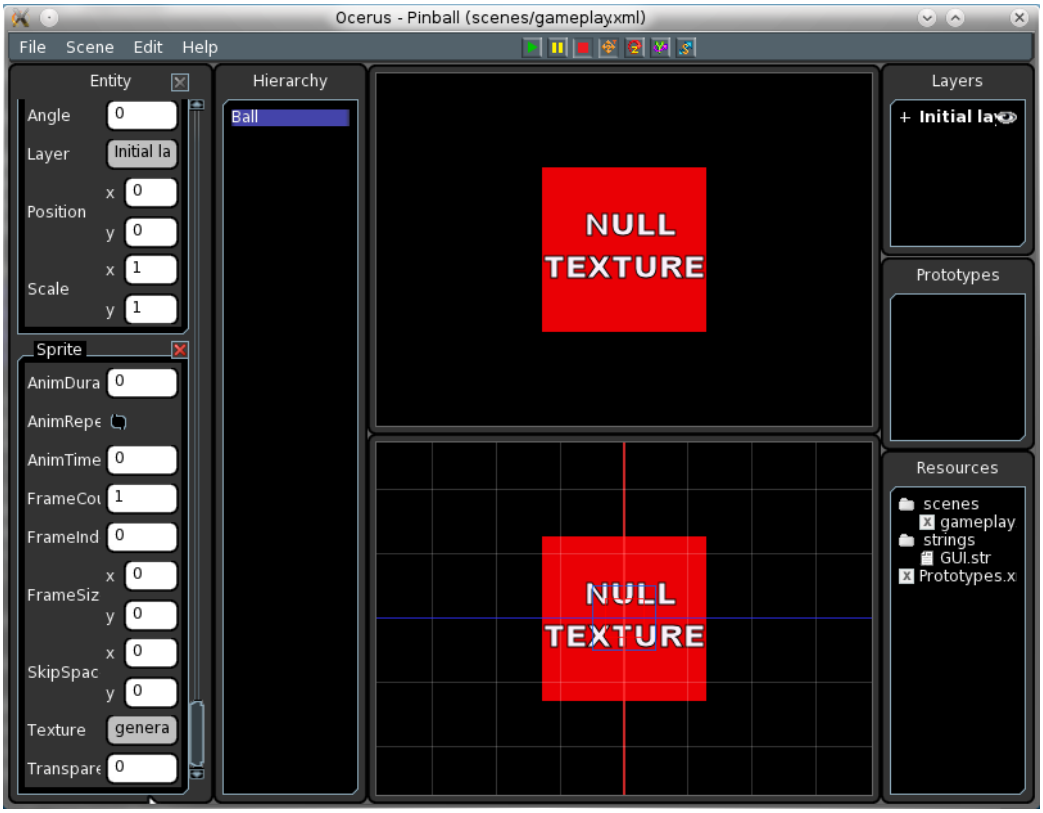


Figure 5: Entity with the Sprite component

# 8 Adding graphics content

Every file and directory that is placed into the project directory hierarchy is visible in the **Resources** view. To put files with your textures into the project, simply copy them somewhere into the project directory. When you add a file into the project directory, Ocerus automatically detects the type of the file and creates the suitable type of resource. The type of the resource is represented by an icon next to the resource name in the **Resource** view.

You can either use the `ball.png` texture from the archive provided with this document, or you can create or get any texture you like. Either way, copy it to 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.

# 9 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.

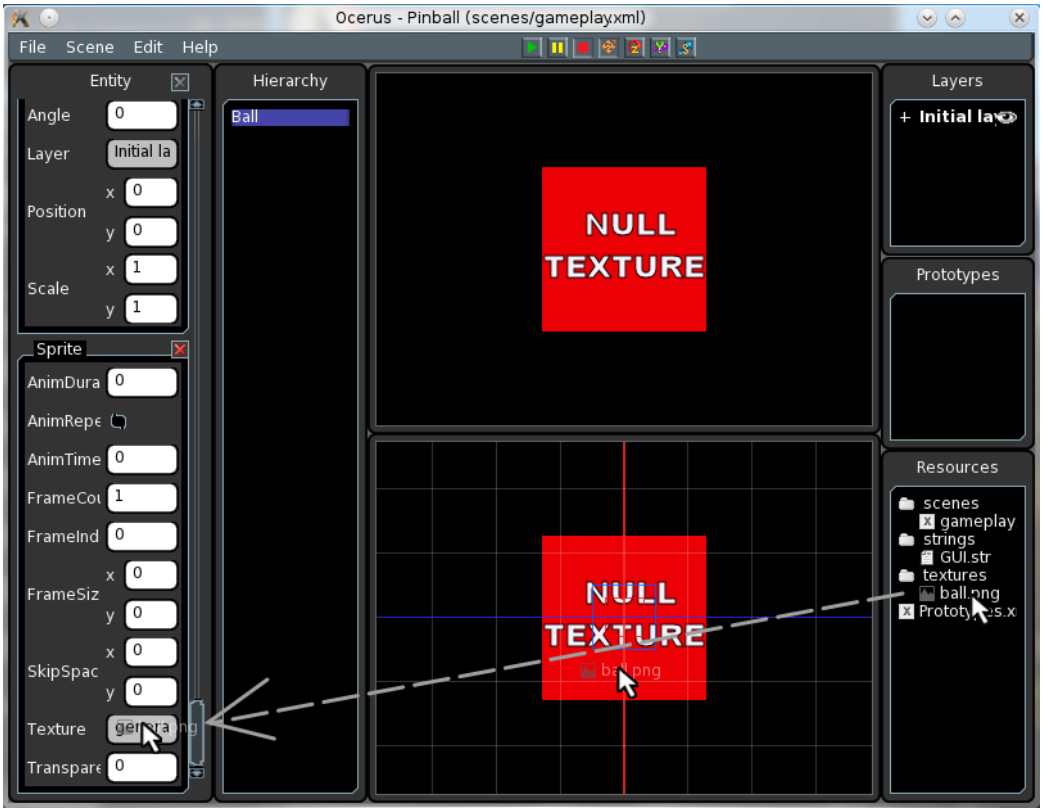


Figure 6: Linking texture to an entity

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

# 10 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 facilitate these operations. There are four editing tools in Ocerus: **move** tool, **rotate** tool, **rotate-y** tool and **scale** tool and they are accessible from the toolbar right from the main menu (see figure 7). You can switch among these editing tools by clicking the corresponding icon, which becomes pressed.

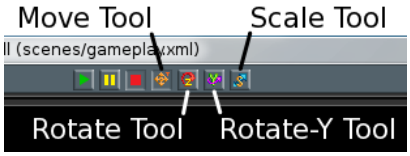


Figure 7: Editing Tools

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 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).

# 11 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 physic 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 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 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 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 according to their collision polygons. So to really see the **PolygonCollider** in action, we have to add physics.

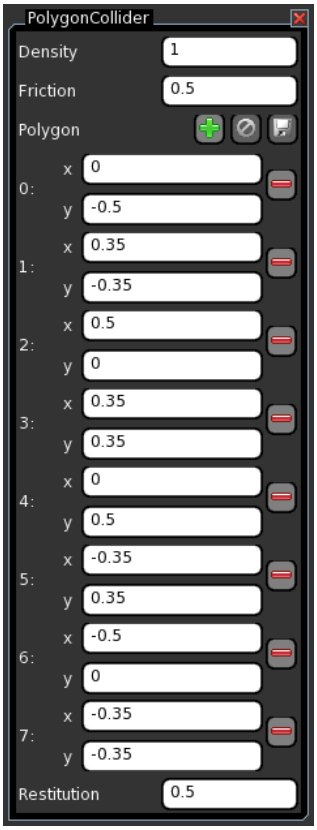


Figure 8: Defining octagon collision polygon

# 12 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. These components are **StaticBody** and **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 gravitation force and other forces. Our ball entity is an example of 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 attracted straight down. It means that as soon as we start our so far created game, our ball would infinitely fall down. Let's create some barriers in our world.

**Task 1.** Create a new entity called `Platform`, add **Sprite** component

and use the `steel.jpg` file from the archive as the texture. Add the **PolygonCollider** component and set 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 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.

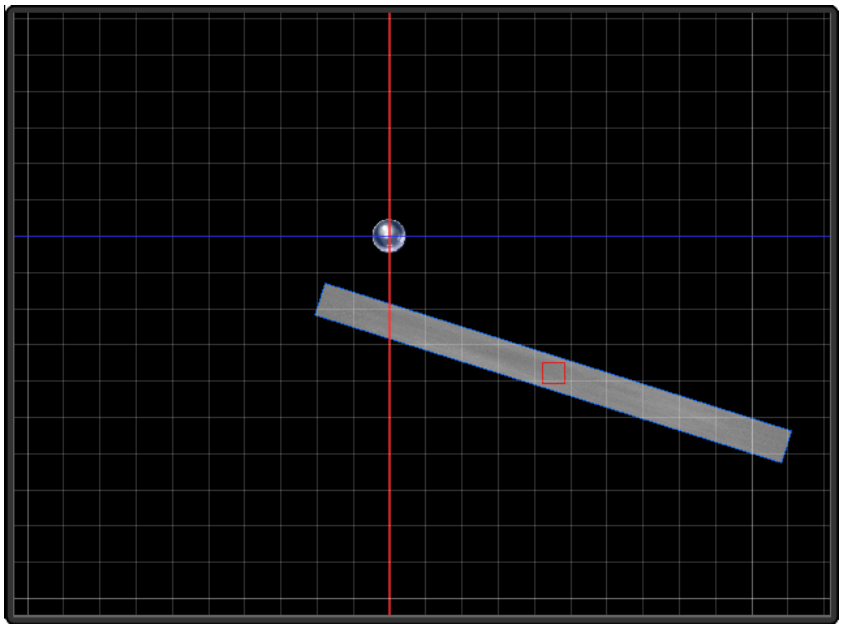

Figure 9: Task 1

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

# 13 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. During the editing you can arrange the initial state of your scene and when you feel like testing everything, you can animate the scene by starting the action. Action is controlled by the action toolbar (see figure 10) that 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 edit tools

and edit entity properties. Changes that you make will be discarded as soon as you stop the action however.

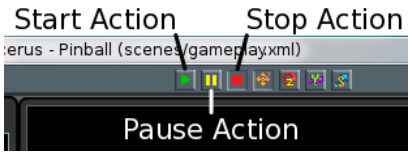


Figure 10: Action Toolbar

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 more smooth. Note that Ocerus does not provide colliders with rounded 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 quite 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 a 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 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.

# 14 Using prototypes

Prototypes are special entity templates. They are not related to a concrete scene, rather they are saved in `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 view to the **Editor viewport**. Drag your new prototype to the viewport a 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.

# 15 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.

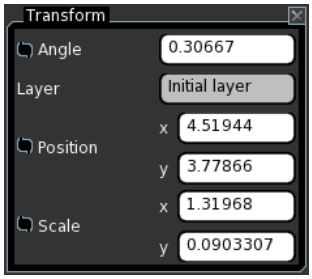


Figure 11: Shared Properties

Select a `Platform` entity from the **Hierarchy** view and again look at the **Entity** view (see figure 12). All properties that are shared have a lock icon on the right and cannot be edited.

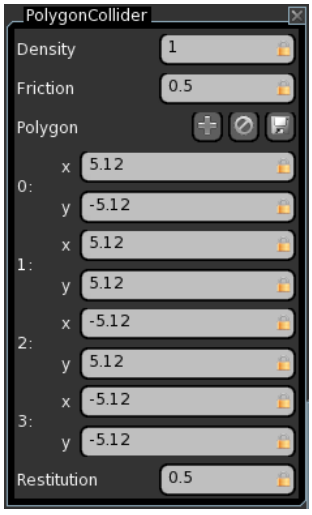


Figure 12: Locked Properties

**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 figure 13. Use the `Platform` prototype to build all barriers. Also create the `Flipper` prototype that would be similar to `Platform` prototype, except that it will use the `yellow.jpg` texture from the archive and it will have **DynamicBody** component instead of **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 from the prototype 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.*

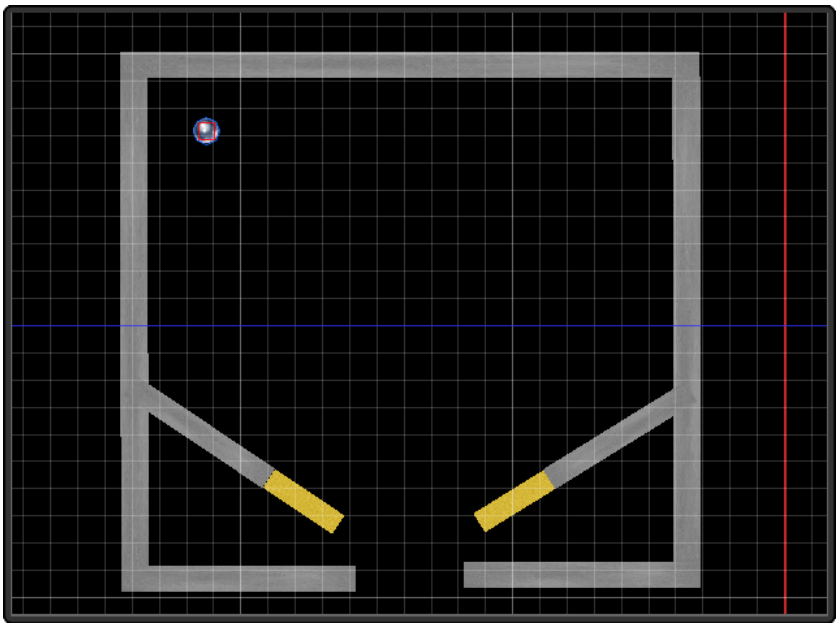

Figure 13: Task 2

# 16 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 file system organization, each and every entity can become a parent of another 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.

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 a 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 be also 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 on the right of the component title in the **Entity** view and move all platform entities under the Platforms entity. Do the same with flippers (see figure 14).

Figure 14: Hierarchy View

To further improve the graphics appearance of our 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 entities' sprite order into account, but now we need to make sure that background stays behind the pinball machine. This can be achieved by using layers.

# 17 Using layers

If you look at the **Layers** view, you can see the `Initial Layer` there. Every scene has to have at least one layer and this one is automatically created. You can click on the plus sign on 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 the another layer and it becomes active. You can also

toggle the layer visibility by clicking the eye icon on the right of the layer.

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

# 18 Scripting

Ocerus engine allows to implement the game behavior using the scripting engine. Although writing scripts is indeed programming, it differs from core Ocerus programming in many ways. First, 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 means 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 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 `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))
    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))
    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 to parametrize 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 another 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 the **Right** key.

# 19 GUI

The last topic that is covered in this user guide is the 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, namely Layout editor and Imageset editor.

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 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 theCEGUI manual to get more info on **Font** files.

The **Imageset** files allow to define your custom images that can be used in GUI. CEGUI allows to put multiple images into a large image file and specify which parts of the large image correspond to particular images. This map is defined in **Imageset** files. In our `Pinball` project, we will define faces of two buttons that allow 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 file.

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.

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.

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 needs 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).

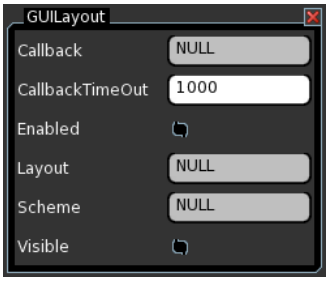


Figure 15: GUILayout component

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. Finally check the **Enabled** and **Visible** checkboxes.