

# PlayCanvas unofficial

only about PlayCanvas



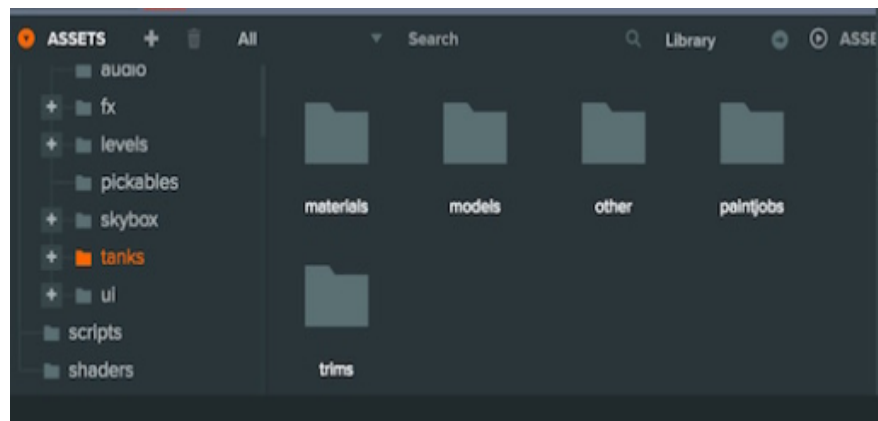
Autor: bunnymq

# TOC

|   |           |
|---|-----------|
| <b>Part I - general</b>                   | <b>4</b>  |
| Chapter 1 PlayCanvas, features            | 5         |
| 1.1 PlayCanvas - what is it?              | 5         |
| 1.2 features                              | 6         |
| Chapter 2<br>assets                       | 10        |
| <b>Part II - editor</b>                   | <b>17</b> |
| Chapter 3<br>UI                           | 18        |
| 3.1 Hierarchy panel                       | 19        |
| 3.2 Resources panel                       | 20        |
| 3.3 Inspector panel                       | 21        |
| 3.4 Menu Panel                            | 21        |
| 3.5 Panel Toolbar                         | 22        |
| 3.6 Viewport                              | 22        |
| <b>Part III - engine</b>                  | <b>24</b> |
| Chapter 4<br>scripting                    | 25        |
| Chapter 5<br>Graphics                     | 31        |
| 5.1 Camera                                | 32        |
| 5.2 Lighting                              | 32        |
| 5.3 PBR                                   | 33        |
| Chapter 6<br>API overview                 | 36        |
| <b>Part IV - Entity - game object</b>     | <b>45</b> |
| Chapter 7<br>Entity, Component,<br>System | 46        |
| <b>Part V - examples</b>                  | <b>54</b> |
| Chapter 8<br>PlayCanvas - examples        | 55        |
| 8.1 After the Flood                       | 55        |
| 8.2 Casino                                | 59        |
| <b>Appendix A</b>                         | <b>64</b> |
| <b>API</b>                                | <b>64</b> |
| <b>Appendix B</b>                         | <b>72</b> |



# Part I - general



# Chapter 1

## PlayCanvas, features

### 1.1 PlayCanvas - what is it?

A game engine created in WebGL, not based, not using 3D threejs library, It was written from scratch, also as a cloud platform for creating games, visualizations, product configurators (e.g. car configurator). You can make very advanced graphics in it, thanks to the capabilities of the engine.

It has an integrated Ammo physics engine. You can use PlayCanvas as a platform with graphical editor and code editor in the cloud or as an engine-only, that is having the project locally on your laptop and using only the engine in the IDE or code editor of your choice (VS Code, Atom, Sublime Text), something like it is implemented in three.js.

With engine-only there is one advantage you can use git and upload to github, in the case of the platform you have to use PlayCanvas' version control system and checkpoints rather than commits like it works in git.

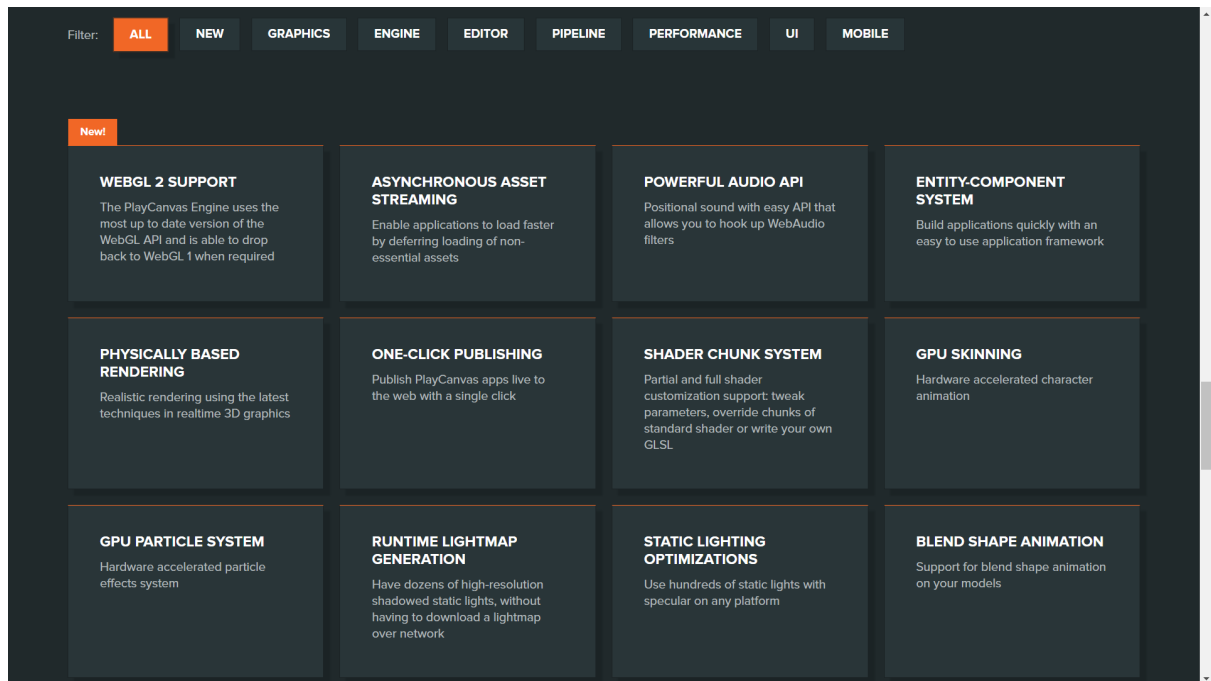
The name looks a bit like it's going to be about classic canvas, HTML5, which is 2D, but yet it's a rich 3D engine, a similar engine to PlayCanvas is Babylon.js, also worth a try, and PlayCanvas I recommend a really cool engine, but honestly only recently there was one drawback before, there was a resource space limit of 100MB if you used the platform. Now the limit is 1GB, so you can keep resources on the platform for free if the total does not exceed 1 GB, to have for example 10 GB or more you need to have a monthly subscription.

The community is not small, the documentation is well written, but there is one drawback: there are no books on the subject, neither in English nor in Polish.

In case of problems you can find a post concerning your problem or you can ask on the forum by creating a new post. It's not that nobody answers your question, you get an answer very fast and even a possible solution to your problem, which is a very strong advantage of PlayCanvas forum. I provide a link to the forum here [PlayCanvas Discussion](#)

Out of curiosity I looked through the engine code, it is really big, although not as huge as it is for Unreal Engine.

## 1.2 features



PlayCanvas has the following features:

WebGL 2 support

Asynchronous resource streaming

audio API

ECS (Entity Component System) - about this in the Entity section

Physics-based rendering (PBR)

System chunk shader

GPU skinning

GPU particle system

Real-time light map generation

shape blending animation

soft shadows and light cookies

Resource importer and manager

Linear graphics pipeline and HDR

Input device API

SDF font renderer

rigidbody physics engine

tools for responsive interfaces

WebVR support

development and testing on a mobile device

resource filtering

real-time scene editing

cubic texture prefiltering

profiler

Texture compression (DXT, PVR and ETC1)

material editor

Cross-platform

WebGL 2 support

Engine uses the latest WebGL API, but is backward compatible with WebGL version one.

asynchronous streaming of resources

Asynchronous, and therefore faster loading of the application, by delaying the loading of less important resources.

audio API

Positional audio allows you to attach WebAudio filters.

ECS (Entity Component System) - about it in the Entity section

Create applications quickly using ECS.

physics-based rendering (PBR)

Bring realism to rendering with the latest real-time techniques in 3D graphics

shader chunk system

Partial and full shader customization: adjust parameters, overwrite standard shader chunks, or write your own GLSL code.

GPU skinning

Hardware accelerated character animation.

GPU particle system

Hardware accelerated particle system

Real-time light map generation

You can have multiple high resolution static lights

Shape blending animation

Support for shape blending animation of models

Soft shadows and light cookies

Choose from multiple shadow algorithms.

Light cookies provide cool effects at a cheap performance cost

asset importer and manager

Import assets: 3D models and animations (FBX, OBJ, DAE, 3DS), textures and HDR

textures, audio files and more

linear and HDR graphics pipeline

Linear and HDR pipeline: gamma correction, tonemapping, support for HDR cubic

textures and lightmaps

Input device API

Keyboard, mouse, gamepad, touchscreen support

SDF font renderer

Convert TTF, OTF to font resources (similar to Unity)

rigidbody physics engine

PlayCanvas' built-in Ammo physics system, which is a port of Bullet, allows for easier

implementation of physics in the game

tools for responsive interfaces

Components to create responsive 2D and 3D interfaces

WebVR support

Support for the latest WebVR standards

Development and testing on a mobile device

Fast iterations using live updates on a mobile device

resource filtering

Search and filter your collection of assets

edit scenes in real time

Collaborate style changes on the fly with Google Docs

Cubic texture prefiltering

Set up image-based lighting (IBL) with just one click of a button

profiler

Displays graphs, real-time performance statistics

Texture compression (DXT, PVR, and ETC1)

One-click texture compression

material editor

Quickly adjust visually visible changes to material parameters using the editor



multi-platform

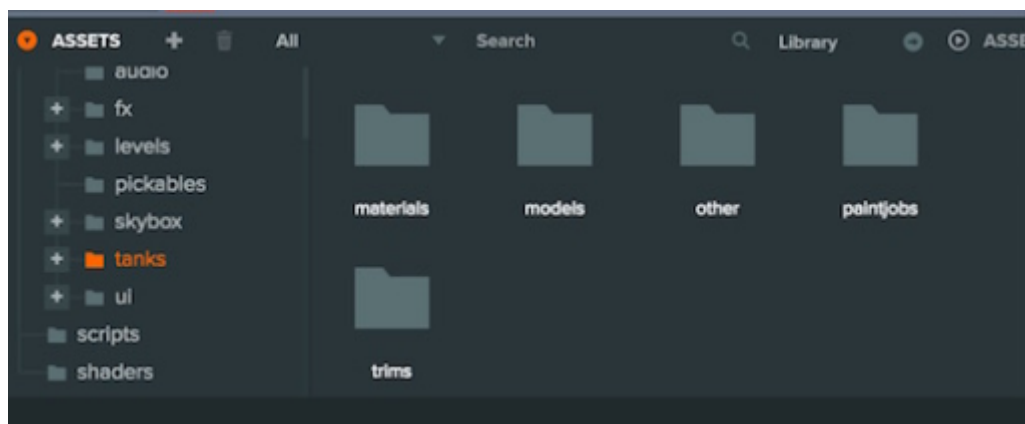
Run the editor on any device: desktop, laptop, tablet, smartphone

As you can see PlayCanvas has a lot of functionality.

I will now move on to discuss the assets.

# Chapter 2

## assets



Assets can be of various types, e.g. model, animation, images for textures (.png,.jpg) and audio.

Below I discuss all types of resources available in PlayCanvas:

material

- Phong

- physical

texture

model

animation

cubemap texture

HTML

audio

CSS

shader

font

sprite

prefab (in pc under the name template)

Wasm module (Wasm module, WebAssembly module)

## 2.1 material

In general, the material defines surface properties such as color, gloss, etc. For exactly what this refers to please refer to your computer graphics textbooks.

In PlayCanvas, a material is one type of resource.

It has 2 subtypes: Phong and Physical.

### Phong

The Phong shading model is an obsolete item, it is recommended to use the physical model.

You can find more about the Phong shading model here [Phong Material | Learn PlayCanvas](#)

### physical

Physical material represents an advanced, high quality shading model and is therefore recommended for use to achieve impressive results.

Detailed information about physical material properties is available here

[Physical Material | Learn PlayCanvas](#)

The following regions are related to this material: offset and tiling, ambient (related to ambient occlusion), diffuse (diffuse is also called albedo), specular (gives shine), emissive (emits light), opacity (transparency), normals (related to normal map), parallax (related to height map), environment (reflections), lightmap,

## 2.2 texture

A texture is an image that can be assigned to a material.

Below I have highlighted texture maps that are useful when multitexturing to get a more detailed look of the material.

Types of texture maps: ambient occlusion (AO map), cubemap, env map, diffuse map, specular map, emissive map, opacity map, normal map, height map, light map.

More about textures here [Textures | Learn PlayCanvas](#)

## 2.3 model

3D models and animations are created outside of PlayCanvas, exported from Blender, Wings3D, Maya or 3DS Max for example, and imported into PlayCanvas.

It is recommended to use the fbx format for best results and so the model will be converted to glb (i.e. fbx will remain as the source format, but glb will be created as the target format and thus there will be two fbx and glb formats for the model).

More about Models | [Learn PlayCanvas](#)

## 2.4 animation

The animation resource is used to play a single animation on a 3D model.

Full scene formats include animation, for example it is gltf, dae, fbx.

More about Animation | [Learn PlayCanvas](#)

## 2.5 cubemap texture

Cubic texture is a special texture type consisting of 6 texture resources.

It is used as skybox or environment map.

More about cubemap [Cubemaps | Learn PlayCanvas](#)

## 2.6 HTML

The HTML resource contains the HTML code.

To load HTML you need to write a piece of js code like this:

```
this.element = document.createElement('div');
this.element.classList.add('container');
document.body.appendChild(this.element);
this.element.innerHTML = this.html.resource;
```

Now I will quickly describe how the code works.

It creates a div element dynamically, then adds a class called container. Then it hooks that div to <body> and sets the content of your html as a child element for the container div.

This is one way to do it.

Of course you still have to add an attribute with html name (if you named it as this.html in the code, about attributes later), write html code, drag the html file in editor to a place where you can attach either entities or different resources, in this case it is ui under script component, in ui there is an attribute of resource type named html, only then you have HTML content on your page.

## More about HTML

### HTML | Learn PlayCanvas

#### 2.7 audio

An audio resource is a sound file.

### Audio | Learn PlayCanvas

#### 2.8 CSS

A CSS resource contains the CSS code.

CSS style is attached to the page just as in the case of HTML resources, that is, you add an attribute named css, create CSS code, drag the css file in the editor to a place where you can attach either entities or different resources, for example, the ui under the script component, in the ui is just an attribute of the type of resource named css, only then you have the CSS content on your page, and so the applied appearance.

The code to hook up the CSS is a little different than it was with the HTML resource.

I'll show a slightly different way this time:

```
// get asset from registry by id
const asset = app.assets.get(32);

// create element
const style = pc.createStyle(asset.resource || "");
document.head.appendChild(style);

// when asset resource loads/changes,
// update html of element
asset.on('load', function() {
    style.innerHTML = asset.resource;
});

// make sure assets loads
app.assets.load(asset);
```

So this is how the CSS resource is fetched from the resource registry, the element is created, and the resource is loaded.

## 2.9 shader

The shader resource contains GLSL code, you can also upload files with the extension `.vert`, `.frag` or `.glsl`

```
const vertexShader = this.app.assets.find('my_vertex_shader');
const fragmentShader = this.app.assets.find('my_fragment_shader');
const shaderDefinition = {
    attributes: {
        aPosition: pc.SEMANTIC_POSITION,
        aUv0: pc.SEMANTIC_TEXCOORD0
    },
    vshader: vertexShader.resource,
    fshader: fragmentShader.resource
};
```

```
const shader = new pc.Shader(this.app.graphicsDevice, shaderDefinition);
const material = new pc.Material();
material.setShader(shader);
```

The first two lines are about looking for vertices and frags in the shader resource register. Next, a shader is defined with attributes: position and uv. The contents of the shader resource are appended to the vshader and fshader properties, first the vertex shader, then the fragment shader. As the penultimate step, the shader and material are created. Finally, the shader for the material is set.

## 2.10 font

A font resource contains an image with all the characters of the font. It is used to display text.

More about font here [Fonts | Learn PlayCanvas](#)

## 2.11 sprite

A sprite is a 2D graphic, since the book is about creating a game in 3D, the 2D topic is omitted

More about sprite [Sprite | Learn PlayCanvas](#)

## 2.12 prefab (on pc under the name template)

A prefab, which is a resource that contains a part of an entity, allows you to create multiple instances, so it is useful for constructing objects that look the same, e.g. 1000 trees of one type, 10 buttons that look the same, etc. In PlayCanvas, there are no prefab variants yet, i.e. for example there is a base prefab car, and for example I want to have different cars having the same features but different values, e.g. top speed or acceleration.

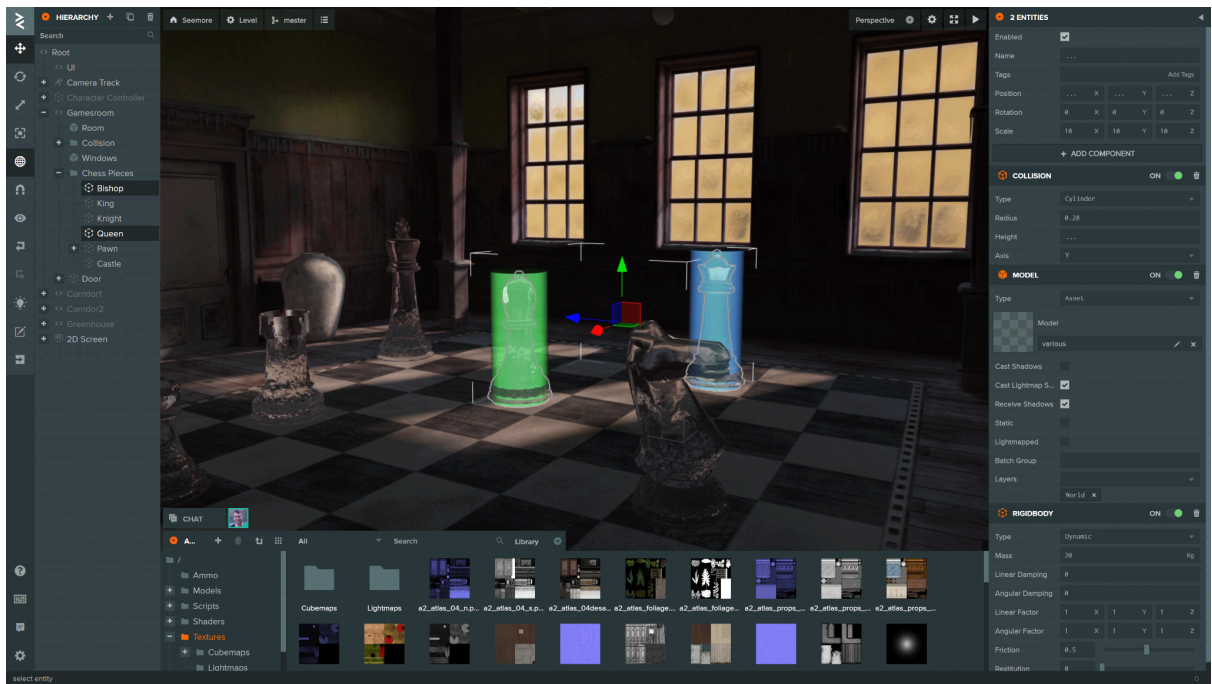
More about prefabs [Template | Learn PlayCanvas](#)

I skipped the topic of the Wasm resource. I think I've covered the possible resources in PlayCanvas pretty well.

Let me move on to the editor part.

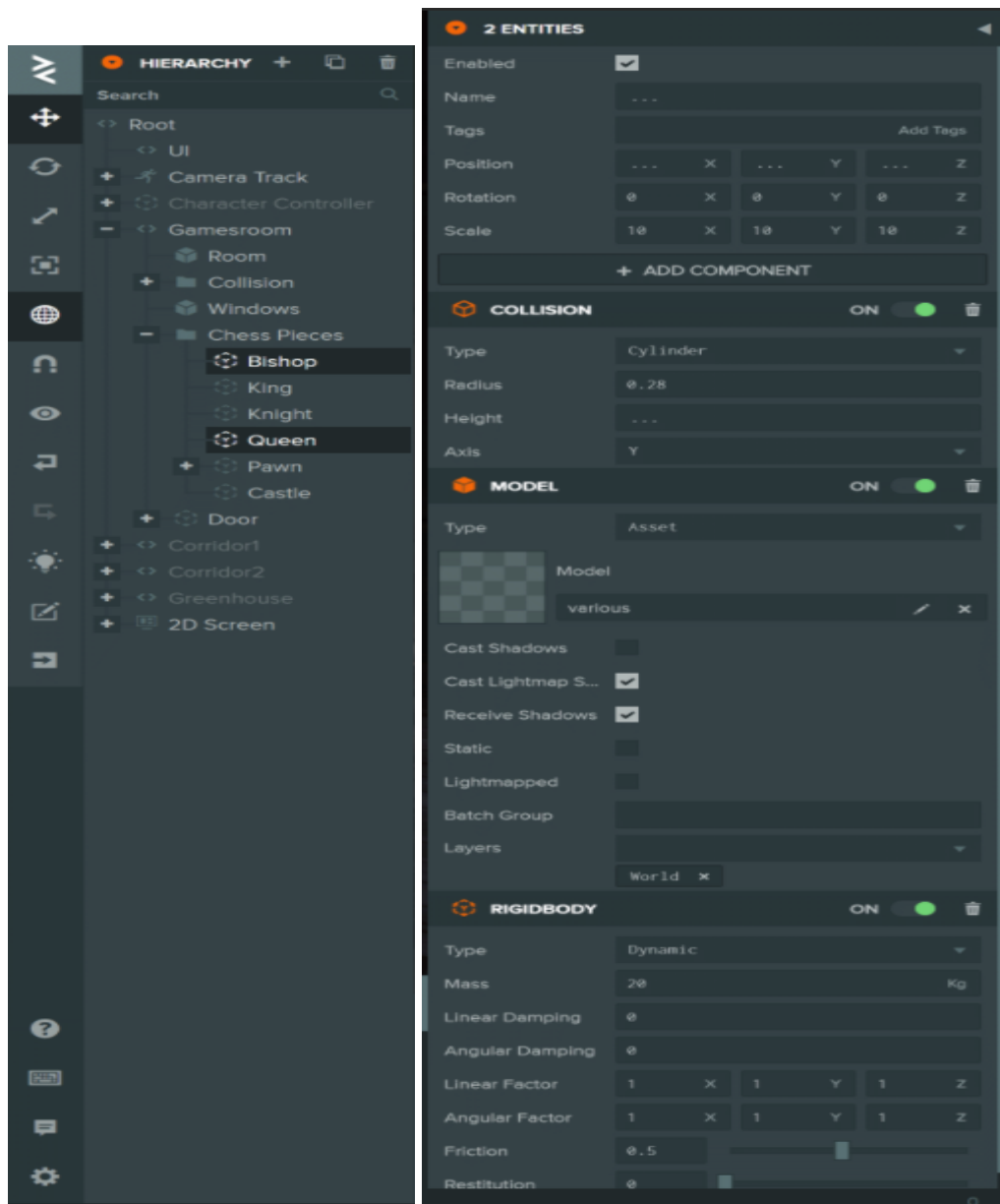


# Part II - editor



# Chapter 3

## UI



## 3.1 Hierarchy panel

The hierarchy panel contains a scene graph, a tree view. This graph consists of a root, by default it is called Root, in this case it is called Main, it can also be called Game.

Main in this case is the parent of entities such as:

Camera, Board and Room Light, Board Folder, Dice1, Dice2, Tokens, Tile Owned, Houses, PropertyEntities, Walls, Cards, Colours, Furniture, New UI, Money, Property Cards, Property Lights and MainEntity.

This is a fragment of the board game Monopoly.

In addition, these entities are the parents (indicated by the plus sign) of other entities and so on.

As you can see, this structure is very complex, so it is important to group objects, e.g. as shown in the figure. Grouping objects is one of the good practices.

Hierarchical structure allows for good organization of game elements.

As a small digression: so look at how this is implemented in other examples of games created in PlayCanvas, go to the PlayCanvas website (you must be logged in to see EXPLORE content, once you are logged in go to explore, you will see various projects there, click on Project next to the particular project.

I chose SWOOP, it is an endless runner game.

This will take you to the next project overview

click on EDITOR,

click on the scene in this case Game and you can see the hierarchy.

I will show some other hierarchies  
like the Space Buggy hierarchy

It's hard to capture on picture the whole developed hierarchy.

I will show one more hierarchy from Accelerally and I will end with hierarchies.

Some projects have locked Project option (e.g. TANX), you can only press PLAY to play the game , picture below.

## 3.2 Resources panel

Resources are best organized in folders, e.g. scripts in scripts, materials in materials, models in models, textures in textures etc.

On the left in the figure you can see the structure of the folders: / is the root and in it there are folders in this case: scripts, Chance, Community Chest, CSS, Furniture, HTML, Money, Other, Properties, Tokens, just like you have it organized in the file system on your operating system.

On the right you can see the folders and files, the folders mentioned above, 2 files: loading.js and redirect.js

Here you can upload your resources, you can filter by categories (here where All is), search for a resource (Search), add a new resource or delete an existing one, you can also enter the PlayCanvas Store.

## 3.3 Inspector panel

Here you can enable / disable the entity (Enabled), name the entity (Name), add tags (Tags), set the transformation: position, orientation (rotation) and size (scale).

Importantly, all these properties are in local space, model space.

The orientation is set using so called Euler angles.

You can also add components (+ add component).

In this case, the added component is the script component which contains the ui.js code.

An entity can have many js scripts attached to it, such as UIHandler, Main, Money, Dice, Movement, Cards, PropertyLight, assets,  
as shown in the figure.

So much for the inspector, there's still a menu and toolbar left to talk about.

I'll move on to the menus.

## 3.4 Menu Panel

The menu can be shown by clicking on the button with the PlayCanvas icon.

The menu lists all the commands you can do in the scene.

Here you can do the following things, add an entity, edit, start the game, get to help, view a list of available scenes, publish the game, burn a map of lights, open settings, set the priority of executing scripts.

Generally this is a shortcut if you can't find the button or can't remember the shortcut key.

## 3.5 Panel Toolbar

Panel toolbar contains the most common commands available in a convenient way.

The most useful is the run button (shortcut ctrl+enter), which starts the game in a new tab, loads the scene you are on and after loading you can test, play.

## 3.6 Viewport

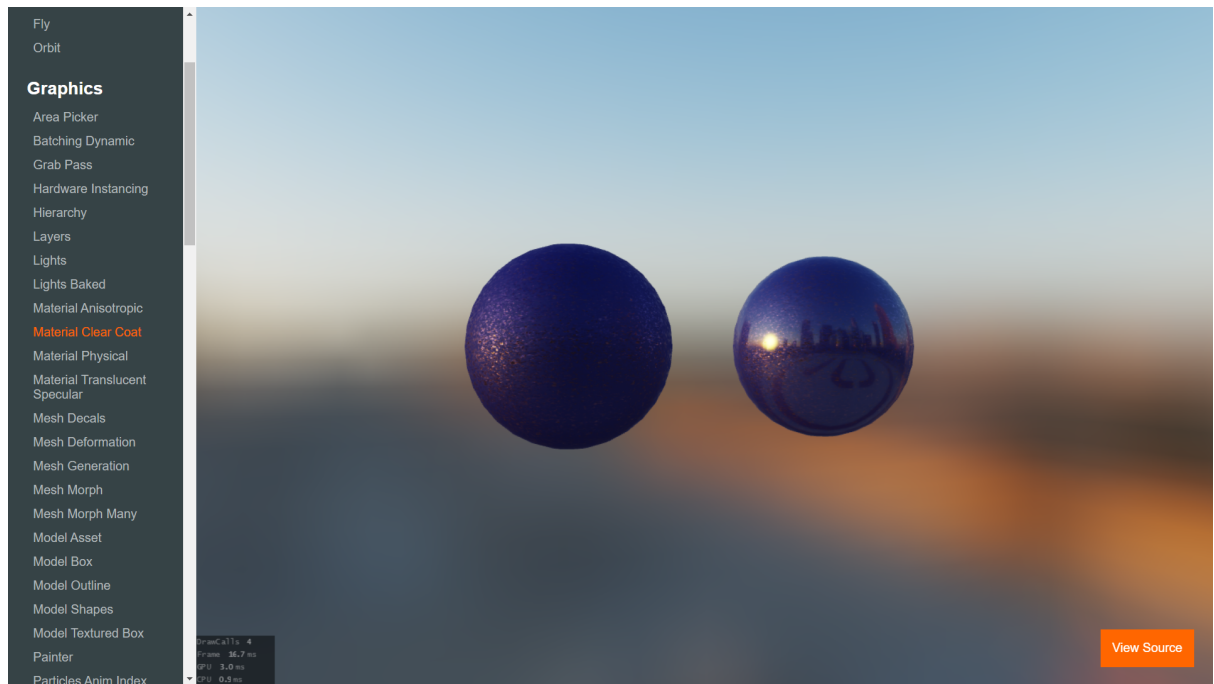
The viewport shows the scene currently displayed. You can move around the scene with the WASD keys and the up, down, left, right arrows. Holding shift accelerates the camera speed, you can view the scene faster this way if the space is large, such as here.

This is a city model, a mod for the game Assetto Corsa, here is a very useful way to quickly view the scene.

Going back to the viewport you can set the camera to perspective or orthographic, in the case of ortho there are left, right, top, bottom, front, back views (left, right, top, bottom, front, back). You can still change to a view from another camera, such as a tracking camera (if you have set one in the hierarchy). In this example, the additional cameras are SplashCamera and Camera. As a side note, cameras are simply matrices.

Now I will move on to discuss the engine part.

# Part III - engine





# Chapter 4

## scripting

```
1  /*jshint esversion: 6 */
2
3  class Ui extends pc.ScriptType {
4
5      // initialize code called once per entity
6      initialize() {
7          this.initHTML();
8      }
9      initHTML(){
10         this.element = document.createElement('div');
11         this.element.classList.add('container');
12         document.body.appendChild(this.element);
13         this.element.innerHTML = this.html.resource;
14     }
15
16
17     // update code called every frame
18     update(dt) {
19
20     }
21
22 }
23
24 pc.registerScript(Ui, 'ui');
25
26 Ui.attributes.add('html', {type: 'asset'});
27
28 // swap method called for script hot-reloading
29 // inherit your script state here
30 // Ui.prototype.swap = function(old) { };
31
32 // to learn more about script anatomy, please read:
33 // http://developer.playcanvas.com/en/user-manual/scripting/
```

### 3.1 initialize and update

**initialize()**

The initialize method refers to initialization that is performed only once for the entity to which the script has been added after the application loads. You use this method when you want to define variables and constants for a particular instance of the script,

e.g. `this.speed`, where `this` refers to the script, not to the window, and `speed` is variable name available in initialize and update, so you can pass `this.speed` variable from initialize to update.

### **update(dt)**

Update can be implemented by using either time-based (time-based) or frame-based. If you do not use `dt`, which is a delta time (delta time) you are using frame-based animation. The result is a lack of fluidity in the animation.

On the other hand, if you add `dt`, then you get smooth animation based on time, not frames per second (fps). That's why you usually multiply the parameter by `dt`, e.g. `this.speed * dt`.

## **3.2 attributes aka attributes.add()**

With attributes you get the ability to iterate faster, i.e. you are able to experiment with parameters, create and test the game faster, e.g. if you are a designer and not a programmer, you can adjust parameters directly in the editor instead of in code. A concept similar to Unity.

You don't even need to use the `dat.GUI`.

There is no point in using attributes if you are engine-only, because you don't have access to the editor then.

Available attributes:

entity

asset

color

curve

enum

JSON

entity

One way even better than using find (find is generally a slow operation, maybe because it uses recursive depth-finding, DFS) is to refer to the entity outside the script using entity attribute. You give the entity name, its type, in the example below I refer to the car to have the information about the speed of the car in the ui. Let's assume that the car entity has a CarController script, then I can access the CarController using this car entity.

```
Ui.attributes.add('car', {type: 'entity'});
```

And here you can see the effect of adding the above line and parsing the code (you still need to press Parse).

Note that car and Car differ in that the former refers to the name of an entity attribute and the latter to the name of an entity in the hierarchy, so stay alert!

Or if you want to find another way, e.g. name both of them differently, I leave it to you as, maybe not a challenge, but a micro-task.

You can also use another way and not attach anything, this way are events about which we will talk later.

### **asset**

The asset attribute allows you to refer to a resource in your project, You can specify only the type as a resource, i.e. type: 'asset' or limit it to a selected type of resource: texture, model, material etc.

The purpose of doing this is to minimize possible confusion, if you can't hook up a resource of every type, only the selected one, you are able to make sure that you can drag into a slot for example only textures and not model or material, to do this you still need to add the assetType property and the type of the specific resource:

```
MyScript.attributes.add('texture', { type: 'asset', assetType: 'texture' });
```

## color

The color attribute shows the color picker, You can specify 'rgb' or 'rgba' as the type.

```
MyScript.attributes.add('color', { type: 'rgba' });
```

## curve

The curve attribute specifies a value that changes over time

```
MyScript.attributes.add('wave', { type: 'curve' }); // one curve
```

## enumerations

With the enumeration attribute, you can choose one value from several options.

The enum property is an array of objects, or options.

In the editor, it works like a dropdown, or HTML's <select>.

## JSON

JSON attribute is a new attribute, it allows nesting of attributes, that is suitable for complex game related structures, The example below is about game configuration.

The schema property is key here, which is an array of objects and contains attribute definitions, but this is already quite specific.

These are all possible attributes of a script. I now turn to the concept of events.

### 3.3 Communication - events

Communication can occur as a transfer of parameters from script to script, i.e. between two entities, for example, or concern a global aspect, i.e. related to application events.

This way has one advantage, you don't have to check every frame with conditional if statements.

#### scriptA-scriptB events

As I mentioned earlier, instead of using entity attributes and dragging an entity to a slot, it can be done using script events. When it comes to scriptA to scriptB events, this is a direct method and therefore faster than application events.

Script events use special `fire()`, `on()` and `off()` methods.

`fire()` triggers the event, `on()` is used for listening, and `off()` disables listening.

The `update()` method calls the event from `fire()` and gives the event name, in this case 'move' with the x, y parameters passed from Player to Display.

The Display class receives the x and y parameters, more specifically a callback method called here `onPlayerMove()`. If the player moves, the x and y values will be displayed in the console, `console.log(x, y);`

To get to the player script, you need to properly reference the player entity (`this.playerEntity`), get to the script component, and finally the player script.

In `on()` you pass the 'move' event name from the previous class (i.e. it depends what you write first whether you start with the `fire()` or `on()` part).

Turn off the 'move' with the `off()` method.

This is how you know how communication works between two scripts, here it was Display and Player.

#### application events

The application is the central store in the context of events, you don't need to refer to entities.

In this example, notice the way the event name is written.

You can see that the event name consists of player and move separated by a colon.

So, this way ensures to avoid name conflicts, e.g. if you have more 'move', it is a good idea to use some namespace, e.g. as above.

fire() fires the 'player:move' event with the x and y parameters.

on() is listening for the 'player:move' event, the key here is this.app.on() and this.app.off(), they refer to the application events.

The result of the action is the same, the x and y value will be printed in the console.

You may have already encountered on() and off() in event-driven programming, fire() is a specific method in PlayCanvas, or at least its name.

I will now move on to graphics, which is the strong point of this engine.

# Chapter 5

# Graphics

PlayCanvas has an advanced rendering engine.

It uses the WebGL API underneath to draw primitives (lines, points, curves, etc.).

In this chapter, I will introduce graphics elements such as camera, lighting, physics-based rendering (I will pay special attention to PBR materials).

I do not focus attention on static and skinned mesh rendering.

## 5.1 Camera

There is no such thing as a camera in WebGL, it is simply a matrix.

But thanks to the game engine, we have access to an implemented camera.

The camera is an object that is responsible for displaying an image of the scene on the screen.

It has various properties: field of view, near and far clip planes etc., but that is a topic about CameraComponent, which can be found in the further section on components.

There are 2 types of camera projections in PlayCanvas: perspective and orthogonal.

Perspective is where objects farther away are smaller and closer objects are larger, so it gives the impression that the image is in 3D.

You had a chance to see perspective when I showed the mod with the city model.

Orthogonal camera works well in 2D or 2.5D games, also called pseudo3D (isometric) figure below.

## 5.2 Lighting

Under the hood of lights are shaders (shading programs), which means that the lighting is simulated in such a way that the color of each pixel is calculated based on the properties of the surface material and light sources. In other words, based on the adopted shading model, which is a certain mathematical model.

The lighting can be dynamic or hidden in the light map.

As far as the lights are concerned, we can choose from the following types of sources: directional, point and spot.

Directional are associated with incident sunlight, meaning they only have a direction.

Spot lights emit light from one point in all directions, which simulates a light bulb.

Spot lights, or reflectors, emit light in a similar way as spot lights. They differ in that the light is confined to a cone shape, which is why they produce such an effect, so they are useful for simulating a flashlight or car headlights.



## 5.3 PBR

Here I will cover a more advanced topic on physically based rendering, which is very popular in games. I will focus on

PBR materials, because in general a book could be written about PBR, and even one such there is a very good one, the link to it is here

Physically Based Rendering: From Theory to Implementation .

Generally PBR consists of: diffusion, specular, energy conservation, metalness, Fresnel term and microsurfaces.

These concepts are shown below.

### Diffuse

Diffuse refers to how the simulated light interacts with the material. As the name implies, the light is scattered in all directions.

### specular

Light that is reflected from a surface to produce a shiny effect.

### Energy conservation

The sum of scattered and reflected light must not be greater than the total light incident on the material. This means that if the surface is highly reflective the effect will be low color dispersion.

### Metalness

Gives the appearance of a material as if it were made of metal.

### Fresnel

The Fresnel effect, the angle at which you look at a surface affects the extent to which the surface shows reflections.

### Microsurfaces

Microsurfaces are related to glossiness, they work on a similar principle as normal maps but on a micro scale, hence microsurfaces. They define whether the surface should be rough or smooth.

The above concepts apply to PBR in general. Now I will present how it is for physical materials.

So yes in PlayCanvas you can use 2 methods called workflow: Metalness and Specular.

When you check Use Metalness, you are using... Metalness.

Otherwise, if you uncheck this option, you use Specular.

Both methods give the same results. It really depends on your preference.

Metalness is related to a value between 0 and 1 (0 - no metallicity, 1 - metal) or a metalness map that will decide which areas of the material will be metallic.

Specular is associated with a specular value or specular map that will determine the color and intensity of the reflected light.

When it comes to PBR materials, the primary component is the diffuse color or albedo.

This is simply the color of the material expressed in RGB (red, green, blue).

You can also use a diffuse map to, for example, assign a material a texture instead of a color.

Suppose for example you are making a game similar to Gran Turismo, and inspired by the licenses you want to have the medals or cups in gold, silver and bronze.

The solution to this is to look for color values on the internet:

As you can see in the table, the first color is gold, and a normalized color value from 0 to 1 and 0 to 255 is given.

The second color is silver, and the third color is maybe not bronze, but quite similar.

And so you have cups in those colors.

It was about metalness and specular, but not yet about glossiness.

The glossiness parameter is used in metalness and specular, as you may have noticed in the workflow picture.

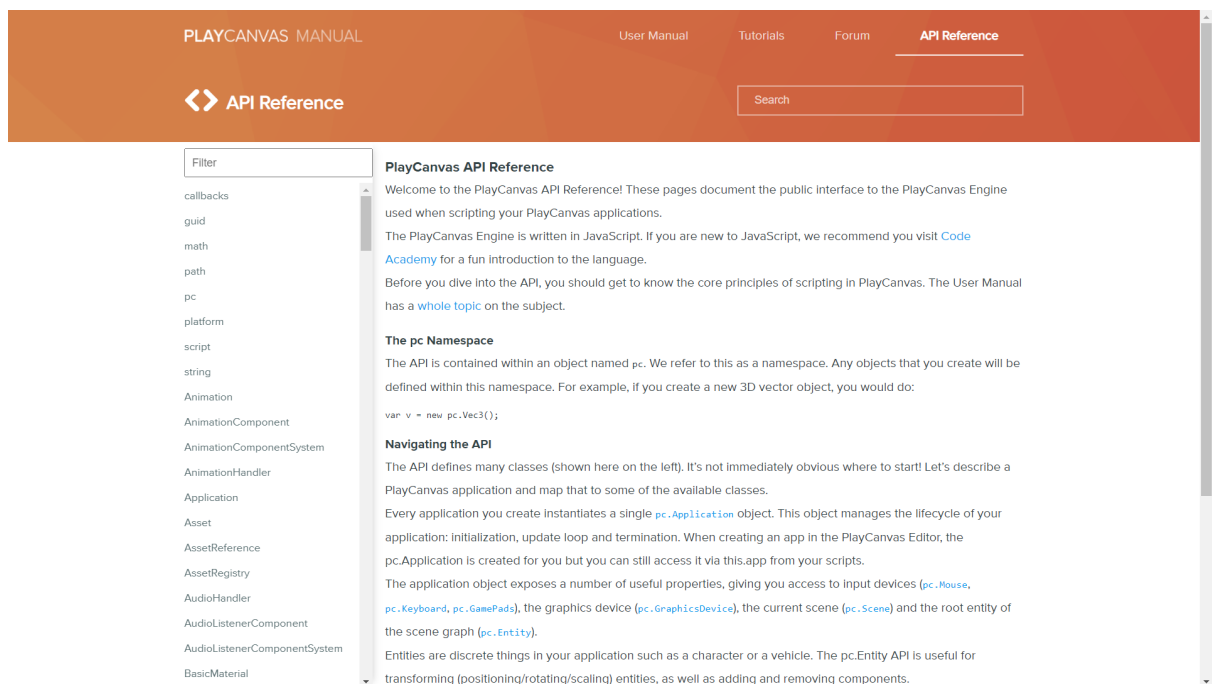
Glossiness determines how smooth the surface of the material is, or the degree of glossiness, which ranges from 0-100. You can also use the glossiness map.

In the picture above it looks a bit like a bauble on a Christmas tree, that's how you can associate it.

So much for the terms metalness, specular and glossiness.

# Chapter 6

## API overview



Here we will analyze such classes as: Application, GraphNode, Entity.

There are also classes about the component, I will call it collectively xComponent (eg. AnimationComponent), where x is one of the given components:

Animation, Audio Listener, Button, Camera, Collision, Element, Layout Child, Layout Group, Light, Model, Particle System, Rigid Body, Screen, Script, Scrollbar, Scrollview, Sound, Sprite.

There are also systems of these components, or xComponentSystem, where x is one of the above components (e.g. AnimationComponentSystem).

Together, this structure (Entity + Component + ComponentSystem) creates something called ECS (Entity Component System). About components and component systems later.

And just like that I've gone through most of the PlayCanvas API, the rest of the classes can be found in Appendix A, and the most up to date list of classes can be found here [PlayCanvas API Reference](#) .

I will now move on to the Application class.

## 6.1 Application

Special attention should be paid to the basic app object of type Application. It is very specific in that it itself contains objects of type:

AssetRegistry, Scene, Keyboard, Mouse, Touch, Gamepad.

One more thing, pc is a namespace (short for PlayCanvas).

pc.app

assets - AssetRegistry

scene

root - Entity

graphicsDevice - graphic device (GraphicsDevice)

input:

keyboard Keyboard

mouse Mouse

touch - mobile device Touch

gamepad Gamepad

Other objects, properties, methods available here [Application | PlayCanvas API Reference](#)

## 6.2 GraphNode and Entity

### 6.2.1 GraphNode

With the GraphNode class you can manipulate entities: add child entities, retrieve / set orientation with Euler angles or quaternions, retrieve / set position in world or local space, retrieve / set scale only in local space, but also allow entity to look at a given point (lookAt()). There are a few other methods not covered in this class.

### 6.2.2 Entity

The Entity class extends / inherits from GraphNode, so it contains inherited methods such as: addChild(), get/set[Local]Position/Rotation(), getLocalScale(), lookAt(), get/set[Local]EulerAngles(), translate[Local](), rotate[Local]().

addChild(node)

Adds a child element, in this example it is an entity

```
const e = new pc.Entity(app);  
this.entity.addChild(e);
```

First an entity object is created and then this child element is added to the parent element (this.entity)

The most common way to set up a tracking camera, then the character/vehicle is the parent entity and the camera is the child entity. This is also a way to dynamically group objects, e.g. parent entity city, child entity building.

Position, orientation and scale can be set with three numbers x, y, z or one vector Vec3

**getEulerAngles()**

gets the Euler angles in world space

```
const angles = this.entity.getEulerAngles(); // [0,0,0]  
angles[1] = 180; // rotate the entity around Y by 180 degrees  
this.entity.setEulerAngles(angles);
```

As above, it retrieves to later rotate the entity around the Y axis by 180 degrees

### **getLocalEulerAngles()**

takes Euler angles in local space

```
const angles = this.entity.getLocalEulerAngles();  
angles[1] = 180;  
this.entity.setLocalEulerAngles(angles);
```

gets to later rotate the entity around its own Y axis by 180 degrees

### **getLocalPosition()**

gets the local position

```
const position = this.entity.getLocalPosition();  
position[0] += 1; // move the entity 1 unit along x.  
this.entity.setLocalPosition(position);
```

gets to later move the entity one unit along x.

### **getLocalRotation()**

gets the local orientation

```
const rotation = this.entity.getLocalRotation();
```

### **getLocalScale()**

takes a local scale

```
const scale = this.entity.getLocalScale();  
scale.x = 100;  
this.entity.setLocalScale(scale);
```

gets to set size x to 100 units later

### **getPosition()**

gets a position in the world space

```
const position = this.entity.getPosition();  
position.x = 10;  
this.entity.setPosition(position);
```

gets to later set the x position to 10 units

### **getRotation()**

gets orientation in world space

```
const rotation = this.entity.getRotation();
```

### **lookAt(x, [y], [z], [ux], [uy], [uz])**

allows an entity to look at another entity, i.e. at a given point

```
// Look at another entity, using the (default) positive y-axis for up  
const position = otherEntity.getPosition();  
this.entity.lookAt(position);
```



```
// Look at the world space origin, using the (default) positive y-axis for up
this.entity.lookAt(0, 0, 0);
```

You can also set the entity to look at point 0, 0, 0

### **rotate(x, [y], [z])**

Rotates an entity in world space

```
// Rotate via 3 numbers
this.entity.rotate(0, 90, 0);
// Rotate via vector
const r = new pc.Vec3(0, 90, 0);
this.entity.rotate(r);
```

### **rotateLocal(x, [y], [z])**

Rotates an entity in local space

```
// Rotate via 3 numbers
this.entity.rotateLocal(0, 90, 0);
// Rotate via vector
const r = new pc.Vec3(0, 90, 0);
this.entity.rotateLocal(r);
```

### **setEulerAngles(x, [y], [z])**

sets Euler angles in world space

```
// Set rotation of 90 degrees around world-space y-axis via 3 numbers
this.entity.setEulerAngles(0, 90, 0);
// Set rotation of 90 degrees around world-space y-axis via a vector
const angles = new pc.Vec3(0, 90, 0);
```

```
this.entity.setEulerAngles(angles);
```

### **setLocalEulerAngles(x, [y], [z])**

sets Euler angles in local space

```
// Set rotation of 90 degrees around y-axis via 3 numbers
this.entity.setLocalEulerAngles(0, 90, 0);
// Set rotation of 90 degrees around y-axis via a vector
const angles = new pc.Vec3(0, 90, 0);
this.entity.setLocalEulerAngles(angles);
```

### **setLocalPosition(x, [y], [z])**

sets the local position

```
// Set via 3 numbers
this.entity.setLocalPosition(0, 10, 0);
// Set via vector
const pos = new pc.Vec3(0, 10, 0);
this.entity.setLocalPosition(pos);
```

### **setLocalRotation(x, [y], [z], [w])**

sets the local orientation

```
// Set via 4 numbers
this.entity.setLocalRotation(0, 0, 0, 1);
// Set via quaternion
const q = pc.Quat();
this.entity.setLocalRotation(q);
```

### **setLocalScale(x, [y], [z])**

sets the local scale/size

```
// Set via 3 numbers
this.entity.setLocalScale(10, 10, 10);
// Set via vector
const scale = new pc.Vec3(10, 10, 10);
this.entity.setLocalScale(scale);
```

### **setPosition(x, [y], [z])**

sets the position in world space

```
// Set via 3 numbers
this.entity.setPosition(0, 10, 0);
// Set via vector
const position = new pc.Vec3(0, 10, 0);
this.entity.setPosition(position);
```

### **setRotation(x, [y], [z], [w])**

sets orientation in quaternions in world space

```
// Set via 4 numbers
this.entity.setRotation(0, 0, 0, 1);
// Set via quaternion
const q = pc.Quat();
this.entity.setRotation(q);
```

### **translate(x, [y], [z])**

Moves an entity by x units in world space

```
// Translate via 3 numbers
```

```
this.entity.translate(10, 0, 0);  
// Translate via vector  
const t = new pc.Vec3(10, 0, 0);  
this.entity.translate(t);
```

Here it moves 10 units in world space

**translateLocal(x, [y], [z])**

Moves an entity by x units in local space

```
// Translate via 3 numbers  
this.entity.translateLocal(10, 0, 0);  
// Translate via vector  
const t = new pc.Vec3(10, 0, 0);  
this.entity.translateLocal(t);
```

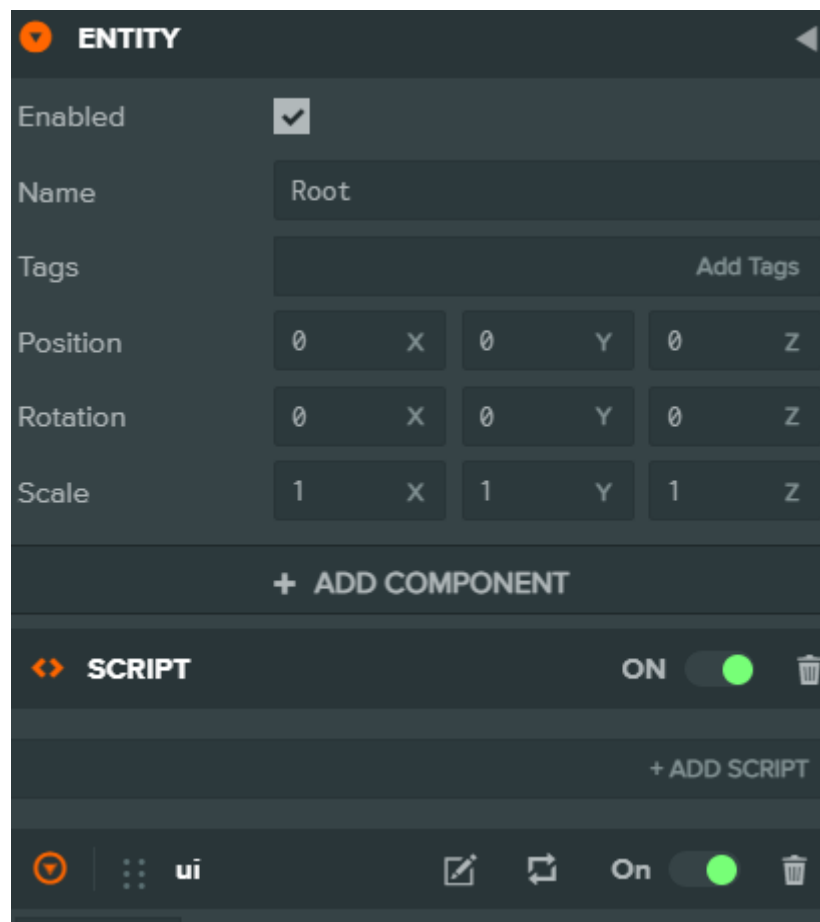
Here it moves 10 units in local space

# Part IV - Entity - game object

# Chapter 7

## Entity, Component, System

### Entity



An entity is a game object (in the Unity engine it is called `GameObject`) which is a container for components, thanks to it you can set the transformation of the object (position, orientation, size). You can also hide / show the object (property `Enabled`).

The most common case is hiding one screen, e.g. the main menu, and showing another, e.g. with options or game settings. Nothing stands in the way of an entity being empty and not containing any components, e.g. if you want it to be a point.

Suppose you are making a configurator and want to have hotspots. A good way to do this is to create empty entities that will represent a point.

Or another example: you want to have a camera transition from point A to point B, empty entities will work well.

Note: if you create a new entity, name it camera, it is not a camera because attaching a CameraComponent to it (either from the editor on the platform editor or engine-only) will make the entity a camera.


Okay, but what if an entity contains more than one component, what then is it? :)

You can still use an entity e.g. when you want to have a script, but not in Root.

The process is as follows: you create a new entity, add to it the script component and add the js code, you name this entity according to the code name.

The following picture may convey what I mean.

Again, I'm using an example from the game Monopoly. So you create an entity that's called UI, you add a script component to it, you add code like uiManager.js, trade.js, setup.js.

You can still see in the figure that the UI descendant components, i.e. Main, Main Menu, etc., have been done similarly, as indicated by the  'tags' icon.

This is one such trick to allow better organization in the hierarchy, above all it is more understandable than pushing all the scripts into Root.

In summary an entity can contain components or not, in which case it is an empty entity.

## **Components**

A component gives an entity a certain behavior, as I said, for an entity to be a camera it must have a camera component. There are other components as well.

Generally components are a certain way of designing software. Components in this case are better than class approach, because with components it is easier to manage things in the game, because they are more modular and smaller than classes, that's it in a nutshell.

Components I will not analyze in detail. Below is a list of 18 components.

Components (18):

- Animation
- Audio Listener
- Button
- Camera
- Collision
- Element
- Layout Child
- Layout Group
- Light
- Model
- Particle System
- Rigid Body
- Screen
- Script
- Scrollbar
- Scrollview
- Sound
- Sprite

Animation

I'll start with the animation component, it determines which animation resources are taken into account.

Audio Listener

The location of the audio listener is given in this component. It refers to



positional audio in 3D. An example can be found here

<https://playcanvas.github.io/#sound/positional.html>.

## Button

Creates a button UI,

## Camera

Adds a camera to the entity that displays the scene from within the entity. The main properties

of the component are the projection type: perspective or orthogonal, projection

(perspective

or orthographic), field of view in degrees, clipping planes

near and far (nearClip, farClip), i.e. at what distance the camera can see from near and far. Other properties can be found here [Camera | Learn PlayCanvas](#)

## Collision

The Collision physical component is useful for collision detection. The available types are

collision shapes are: box, sphere, capsule, mesh,

compound (composed of other basic shapes: box, sphere, etc.), cone

cylinder. If an entity has only a collision component, it is called a trigger, which

works well for detecting when to open a door in the game or when you have reached the finish line. Collision is most often used with rigidbody which is discussed later.

One more thing about collision, because there will be a situation that you want to move collision shape, but the problem is that you also move the 3D model at the same time.

The trick is to create a parent entity, throw there the collision component, and give the model as a child entity. As a result, you can move the collision shape without

moving the 3D model.

## Element

The Element component is a child element for the Screen component.

It builds the user interface with images or text, so you can choose the type: Image or Text.

## Layout Child

Layout Child refers to the ability to override Layout Group parameters.

## Layout Group

In Layout Group, you can set the orientation of the UI elements to Horizontal or Vertical with the Orientation property, which takes Horizontal or Vertical values.

An important property is Wrap, by activating this property you get a grid, that is elements located in rows and columns.

## Light

As you already know light can have a type: directional, spot or reflector.

In PlayCanvas this is defined by the Type property with possible options: Directional, Point or

Spot. You can give the light a color and intensity with the Color and Intensity properties.

## Model

Model, with it you can display primitives: box, capsule, cone, cylinder, plane (plane, I call it floor), sphere. In addition to primitives, you can also display a 3D model (in .glb, .fbx etc), to do this you need to set Type as Asset. The Model component has a Model property where you can find a slot into which you can drag and drop your chosen 3D model asset.

## Particle System

With this component you can set up a particle system, be it rain or snow.

## Rigid Body

Another physical component is Rigidbody, it is used with the Collision component.

Rigidbody gives entities physical properties such as mass, linear and angular damping, friction, and restitution.

Additionally, Rigidbody has 3 types: Static, Dynamic, and Kinematic.

Kinematic is not affected by any forces, Static is useful for ground, track etc.

Dynamic is used for physics along with forces.

A little tip: Don't forget to set the Reflectivity property to 0 for the Static type if you don't want the object to bounce.

## Screen

Screen is the parent for the Element component. It defines the display area of the component.

## Script

The Script component is the primary component most often used. You can use it to attach scripts in js (i.e. not directly figure below).

As an example, you can disable the Script component if you have attached a script from turn.js and don't want the model to rotate.

As another example, you can also enable this component to activate CarController.js when you want the car to start from the starting line only after the countdown 3,2,1.

There are probably more ways.

## Scrollbar

Scrollbar refers to the scroll control for the following Scrollview component.

## Scrollview

Scrollview specifies the scrollable area.

## Sound

The Sound component controls sound, plays audio resources.

## Sprite

There are two types inside this component: a simple or animated soul. Sprite displays the resources associated with the Sprite.

These are all the possible components in PlayCanvas. I didn't discuss the API of these components, so you can find the rest here

<https://developer.playcanvas.com/en/api/?filter=component>

As for this ComponentSystem, an overview of that can be found below, which is about Systems.

## Systems

Systems, or more precisely component systems in other words managers, and in PlayCanvas ComponentSystems are responsible for managing components. As an example I will set the global gravity for all Rigidbody components, and in this case I will use RigidbodyComponentSystem, but more precisely the ready-made object belonging to app.systems.

### ComponentSystemRegistry systems

The application's component system registry. The Application constructor adds the following component systems to its component system registry:

- animation (AnimationComponentSystem)
- audiolistener (AudioListenerComponentSystem)
- button (ButtonComponentSystem)
- camera (CameraComponentSystem)
- collision (CollisionComponentSystem)
- element (ElementComponentSystem)
- layoutchild (LayoutChildComponentSystem)
- layoutgroup (LayoutGroupComponentSystem)
- light (LightComponentSystem)
- model (ModelComponentSystem)
- particlesystem (ParticleSystemComponentSystem)
- rigidbody (RigidbodyComponentSystem)
- screen (ScreenComponentSystem)
- script (ScriptComponentSystem)

```
scrollbar (ScrollbarComponentSystem)
scrollview (ScrollViewComponentSystem)
sound (SoundComponentSystem)
sprite (SpriteComponentSystem)
// Set global gravity to zero
this.app.systems.rigidbody.gravity.set(0, 0, 0);
```

As you can see, there are other component systems in systems (e.g. animation type `AnimationComponentSystem`). I will spare discussing them, because they all work on similar principle, that is, they involve global settings.

Going back to the rigidbody system example, I access it through `app.systems`, so I have `app.systems.rigidbody`. Then I set the global gravity (or basically no gravity) with `gravity.set(0,0,0)`, so together I get the above line of code, I guess you know how to put it together?

And also `systems` is a registry of systems. Let me end this section and move on to the examples.

# Part V - examples

# Chapter 8

## PlayCanvas - examples

Two technical demos will be analyzed, After the Flood and Casino, I won't discuss the code in detail, because it would surely take hundreds or even thousands of pages, so I'll analyze the Casino demo in terms of hierarchy (and in great detail, not as briefly as in the part about the editor on the hierarchy panel), organization of resources, especially scripts, but also in terms of graphics. Due to the complexity of the After the Flood demo, I will limit myself to a brief review of scripts and graphics.

### 8.1 After the Flood

#### **A brief analysis of graphics and scripting**

After the Flood is a demonstration of the capabilities of WebGL 2 with the cooperation of the PlayCanvas team and Mozilla.

As you can see in the picture, the 3D procedural cloud textures created with Worley-Perlin noise are impressive.

What else does this demo include? It includes procedural water, animated leaves with GPU-side particle system, HDR rendering with multisample antialiasing (MSAA, multisampling antialiasing), hardware PCF for shadows, Alpha to coverage, real-time light map baking and mirroring.

For this project, the js code is not in the scripts folder.

I found these js files:

alphaToCoverage.js, ambient.js, assignLight.js, assignLights.js, changeCamera.js, cloth.js, controller.js, debugShaderExplosion.js, demoSettings.js, dontDrawToDepth.js, finalScene.js, finalTempLamp.js, findIdenticalShaders.js, fly-camera.js, foliage.js, fps.js, hideDistance.js, instancingGroup.js, layerSetup.js, leavesParticle.js, leavesShadows.js, less.min.js, loader.js, loading.js, loadShaders.js, makeBrighter.js, mirror.js, namespace.js, noAlphaTest.js, objExportLib.js, opacityFresnel.js, patchCubemap.js, patchFog.js, placeLeaves.js, post2bloom.js, post2colorCorrection.js, post2grabColor.js, postprocess.js, recordShaders.js, reflectableWorld.js, refractiveMaterial.js, renderDebug.js, replaceMipmappedTexture.js, setLayer.js, setNewLayers.js, shaderBuild.js, shaderComplexity.js, shadowCasterControl.js, singleMesh.js, sky.js, skyboxToggle.js, softParticle.js, specularAA.js, subMirror.js, testExample.js, totalTime.js, tree.js, ui.js, volumelight.js, water.js, wire.js, wireUpdate.js, xlimit.js, zprepass.js

The following list briefly describes each js file:

alphaToCoverage - material property  
ambient - ambient music  
assignLight, assignLights - assign lights  
changeCamera - change camera  
cloth - cloth animation  
controller - character controller  
debugShaderExplosion - explosion shader in debug mode  
demoSettings - technical demo settings  
dontDrawToDepth - does not allow to draw to depth buffer  
finalScene - final scene  
finalTempLamp - final temporary lamp  
findIdenticalShaders - finds identical shaders using edit distance  
fly-camera - flying camera  
foliage - foliage collection  
fps - frames per second  
hideDistance - hides the distance



instancingGroup - instance group needed for batching

layerSetup - layer settings

leavesParticle - animated leaves with particle system on GPU side

leavesShadows - leaf shadows

loader

loading screen

loadShaders

makeBrighter - brightens up street lights

mirror

namespace - global atf object (after the flood)

noAlphaTest

objExportLib - library for obj model export

opacityFresnel - Fresnel transparency (code not active)

patchCubemap - patch cube texture

patchFog - patch fog

placeLeaves - places leaves dynamically

post2bloom - bloom effect

post2colorCorrection - color correction

post2grabColor - grab color

postprocess - postprocessing

recordShaders - save shader cache

reflectableWorld

refractiveMaterial - material refraction

renderDebug - display in debug mode

replaceMipmappedTexture - replaces a mipmapped texture

setLayer - sets a layer

setNewLayers - sets new layers

shaderBuild - build shader

shaderComplexity - shader profiles (random, characters, vertex characters, pixel characters, key, shader link time)

shadowCasterControl - shadow control

singleMesh - combine mesh, batching

sky - procedural clouds

skyboxToggle - toggle skybox

softParticle - soften the particle system

specularAA - mirror antialiasing

subMirror - set offset of the mirror depth

testExample - test example

totalTime - total time

tree - tree instance

ui - user interface, here dynamically switching between menu panels, settings, info etc.

volumelight - spatial light

water - procedural water

wire - wire

wireUpdate - update related to the wire

xlimit - limit of x positions

zprepass - concerning depth buffer

As you can see there is a lot going on in this demo, that's why I resigned from code analysis, I described only js files.

## 8.2 Casino

The Casino demo was created by the PlayCanvas team. It is smaller than After the Flood in terms of amount of code and js files, so I will analyze its hierarchy, supporting myself with screenshots. I will have to split the hierarchy into several screenshots.

Analyzing the hierarchy

I'll start with the main entities and then expand the hierarchy one by one.

As it turns out, the tree structure is flat, only the cameraPath entity contains child entities.

Cameras except Camera4 are most likely not used.

But there is one drawback here, the objects are not grouped.

Generally you can see here such entities as: bar chair, chair, table, tree, poker table, second floor light, slot machine, sofa, bar, reflections, slot machine reflections, roulette table, LOD (slot machine, roulette table, sofa, poker table) detail level, user interface, glow effect, camera path and others. The main entities I counted 117, still the camera path has 12 child entities, which gives a total of 129 entities in the project, whether it is a lot, not much for sure. On this I will end the analysis of the hierarchy.

Analysis of resources

When it comes to the resources in the project there are not many, so the analysis will go quickly and I will limit only to a brief description and show the 3D models.

The main folder contains folders (scripts, json, materials, models, textures) and files (!Cubemap, baseRefl2, ui.html, ui\_css.css).

You can find in the project such 3D models as bar chair, chair, table, tree, poker table, gaming machine, sofa, bar, bar shelf, roulette table, coffee table, among others.

poker table and chairs

slot machine

coffee table and sofa

roulette table

bar and bar shelf

scripts

You can find the following js scripts in the project:

bakeLight.js - burning lights

cull.js - culling the camera

distCull.js - culls the distance

pickReflection.js - picks a reflection

animCamera.js - animates the camera

loadingScreen.js - loading screen

rotateReflection.js - rotates the reflection

fixGloss.js - fixes the glow

occludeWithLM.js - occlude with lightmap (LM - lightmap)

lightProbe.js - light sample

lod.js - level of detail (LOD)  
glow.js - glow effect  
flyCamera2.js - flying camera  
reflectionProbe.js - reflection sample  
overrideLM.js - overrides the lightmap  
renderReflection.js - displays reflections  
fog.js - spatial fog  
tileOffsetWorkaround.js - solves the offset problem  
ui.js - user interface  
foliage.js - foliage  
dontLod.js - don't activate LOD  
cullPrewarm.js - culling  
lightProbeAtEachNode.js - sample the light at each node  
optimize.js - optimize  
bicubicLM.js - bicubic lightmap

short graphics analysis

The demo demonstrates the use of PBR materials mentioned earlier, spatial fog, reflection probe, light probe, engine generated lightmaps, real-time paraboloidal reflections, level of detail 3D LOD models.

In the demo you can switch the camera between kinematic and flying (you can then view the entire scene).

That was it for the graphics.

### **What to pay attention to?**

One more thing you can pay attention to, you can add multiplayer / networking in your game, AI for example with Yuka Yuka | A JavaScript library for developing Game AI, try to integrate PlayCanvas with Tensorflow.js. Add Vue or React, use PCUI, a UI component lib from PlayCanvas PCUI.

You can still build with Apache Cordova to create a mobile game that will be a hybrid app, that way you are able to upload to the Google Store (Google Play) or Amazon App Store.

You have reached the end of the book, thanks for reading it!

I wish you good luck with PlayCanvas, I hope the tricks I described here, and there were a couple of them, will be useful to you :)

Once again, I encourage you to take a look at Physically Based Rendering: From Theory to Implementation , you can also refer to Real-Time Rendering, Fourth Edition 4th Edition .



# Appendix A

## API

### pc

callbacks

guid

math

path

platform

script

string

### Animation

Animation

AnimationComponent

AnimationComponentSystem

AnimationHandler

### Asset

Asset

AssetReference

AssetRegistry

### Audio

AudioHandler



AudioListenerComponent  
AudioListenerComponentSystem

## **Batch**

Batch  
BatchGroup  
BatchManager

## **Component**

Component  
ComponentSystem  
ComponentSystemRegistry

## **Element**

ElementComponent  
ElementComponentSystem  
ElementDragHelper  
ElementInput  
ElementInputEvent  
ElementMouseEvent  
ElementTouchEvent

## **Layout**

LayoutChildComponent  
LayoutChildComponentSystem  
LayoutGroupComponent  
LayoutGroupComponentSystem

## **Model**

Model  
ModelComponent  
ModelComponentSystem  
ModelHandler

## **Morph**

Morph  
MorphInstance  
MorphTarget

## **Script**

ScriptAttributes  
ScriptComponent  
ScriptComponentSystem  
ScriptHandler  
ScriptRegistry  
ScriptType

## **Sound**

Sound  
SoundComponent  
SoundComponentSystem  
SoundInstance  
SoundInstance3d  
SoundManager  
SoundSlot

## **Sprite**

Sprite  
SpriteAnimationClip  
SpriteComponent  
SpriteComponentSystem  
SpriteHandler

## **Texture**

Texture  
TextureAtlas  
TextureAtlasHandler  
TextureHandler

## **Touch**

Touch  
TouchDevice  
TouchEvent

## **Vec**

Vec2  
Vec3  
Vec4

## **Vertex**

VertexBuffer  
VertexFormat  
VertexIterator

## **XR**

XrHitTest  
XrHitTestSource  
XrInput  
XrInputSource  
XrManager

## ***pozostale***

Application  
BasicMaterial

## **bounding**

BoundingBox  
BoundingSphere

## **button**

ButtonComponent  
ButtonComponentSystem

## **camera**

CameraComponent

CameraComponentSystem

## **collision**

CollisionComponent

CollisionComponentSystem

Color

## **contact**

ContactPoint

ContactResult

## **container**

ContainerHandler

ContainerResource

Controller

CubemapHandler

## **curve**

Curve

CurveSet

Entity

EventHandler

## **font**

Font

FontHandler

ForwardRenderer

Frustum

GamePads

GraphicsDevice

GraphNode

Http

I18n

IndexBuffer

## **keyboard**

Keyboard  
KeyboardEvent

## **layer**

Layer  
LayerComposition

## **light**

LightComponent  
LightComponentSystem  
Lightmapper

## **Mat**

Mat3  
Mat4

## **Material**

Material  
MaterialHandler

## **Mesh**

Mesh  
MeshInstance

## **Mouse**

Mouse  
MouseEvent

Node  
OrientedBox

## **particle system**

ParticleSystemComponent  
ParticleSystemComponentSystem

Picker

## **post effect**

PostEffect  
PostEffectQueue

Quat

**ray**

Ray

RaycastResult

RenderTarget

**resource**

ResourceHandler

ResourceLoader

**rigidbody**

RigidBodyComponent

RigidBodyComponentSystem

**scene**

Scene

SceneHandler

**scope**

ScopeId

ScopeSpace

**screen**

ScreenComponent

ScreenComponentSystem

**scrollbar**

ScrollbarComponent

ScrollbarComponentSystem

**scrollview**

ScrollViewComponent

ScrollViewComponentSystem

Shader

SingleContactResult

Skeleton

**skin**

Skin

SkinInstance

StandardMaterial

StencilParameters

Tags

TransformFeedback

**stale**

# Appendix B

## PlayCanvas Examples

### Animacja

- Blend
- Tweening

### Kamera

- First Person
- Fly
- Orbit

### Grafika

- Area Picker
- Batching Dynamic
- Grab Pass
- Hardware Instancing
- Hierarchy
- Layers
- Lights
- Lights Baked
- Material Anisotropic
- Material Clear Coat
- Material Physical
- Material Translucent Specular
- Mesh Decals
- Mesh Deformation
- Mesh Generation
- Mesh Morph
- Mesh Morph Many
- Model Asset



- Model Box
- Model Outline
- Model Shapes
- Model Textured Box
- Painter
- Particles Anim Index
- Particles Random Sprites
- Particles Snow
- Particles Sparks
- Point Cloud
- Point Cloud Simulation
- Portal
- Post Effects
- Render To Cubemap
- Render To Texture
- Shader Burn
- Shader Toon
- Shader Wobble
- Texture Basis
- Transform Feedback

#### Loadery

- Loader Glb
- Loader Obj

#### urządzenia wejścia

- Gamepad
- Keyboard
- Mouse

#### Różne

- Mini Stats
- Multi Application

#### Fizyka

- Compound Collision
- Falling Shapes

- Raycast
- Vehicle

#### Dźwięk

- Positional

#### Spine

- Alien
- Dragon
- Goblins
- Hero
- Spineboy

#### interfejs użytkownika

- Button Basic
- Button Particle
- Button Sprite
- Scroll View
- Text Basic
- Text Canvas Font
- Text Drop Shadow
- Text Localization
- Text Markup
- Text Outline
- Text Typewriter
- Text Wrap
- Various

#### mieszana rzeczywistość (vr, ar)

- Ar Basic
- Ar Hit Test
- Vr Basic
- Vr Controllers
- Vr Hands
- Vr Movement
- Xr Picking

