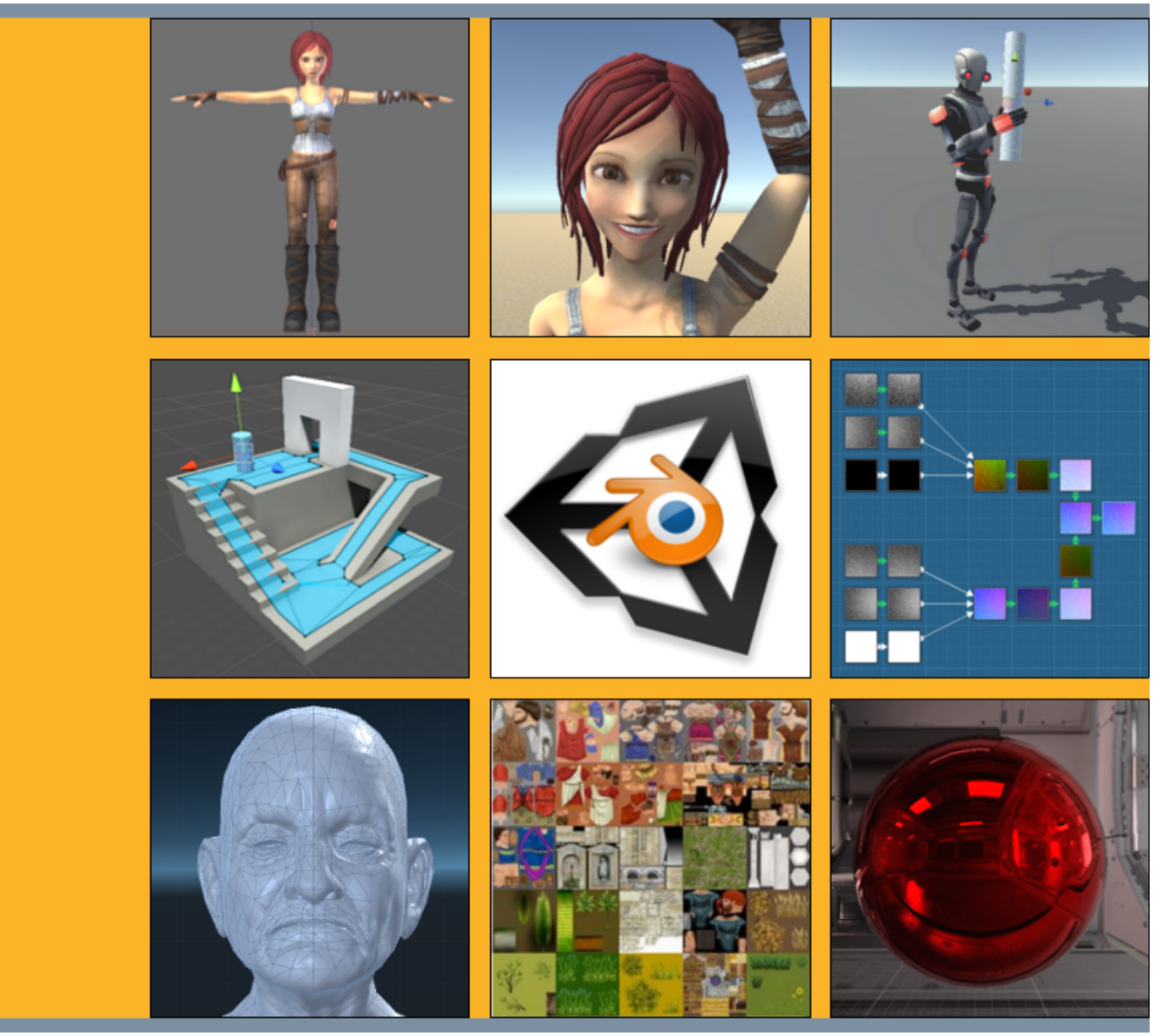
Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

BFH-TI: Bachelor: Informatik: Module 7534: HS 2018

Script

Advanced Game Development



Authors: marcus.hudritsch@bfh.ch,
Marc Wacker, Kaspar Schmid, Camille Zanni

15. Juni 2018

Table of Contents:

1 Module Overview

1.1 Welcome

Welcome to the **Advance Game Development (BTI7534)** module at the *Berne University of Applied Science*. It is the follow-up module of the elective module **Game Development with Unity 3D (BTI7572)** in the spring semester.

The predecessor module is ***mandatory*** for this course because you learn many basics there about Unity that you will use here.

If you did not take the predecessor module (BTI7572) you must show before the first week that you have done the checkpoints 1-4 of from the predecessor module. Please contact marcus.hudritsch@bfh.ch.

1.2 Version History

15.06.2018: Moved various chapters to *Additional Unity Documentations*

1.3 Workload

You will get 2 ECTS credits for this module. For a semester with 16 weeks this will lead to the following weekly workload:

NO. of credits:	2
Workload per credit:	30h
Total workload per semester:	60h
Weeks in semester:	16
Workload per week:	**3.75h**
Contact hours per week:	1.50h
Self-study hours per week:	2.25h

1.4 Time Table & Grading

1.4.1 Tutorials

The semester structure is roughly the same as in the previous module with around ⅔ lectures where we teach you in a tutorial style and check your skills at 2 milestones in week 5 and 10 of the semester.

1.4.2 Project

The last third is a self-defined project that you work on in a group of max. 2 students.

See also the chapter about the Advanced GameDev Project.

1.4.3 Checkpoints

The chapters that are checked at the checkpoints are marked in the script with a green checkmark.

Checkpoint 1: Animated Box
Checkpoint 2: Waving Sintel
Checkpoint 3: Idle-Walking Controller
Checkpoint 4: Blend Tree
Checkpoint 5: Walk, Run & Jump Controller
Checkpoint 6: Blend Shapes
Checkpoint 7: Unity IK
Checkpoint 8: Patrolling AI

1.4.4 Semester Schedule HS19

19. September:	1) Introduction, Ch.2.2.2.1 Create a simple animated box
26. September:	2) Ch.2.2.3 High-level Rigging as exercise (except 2.2.3.4) until checkpoint 2
3. October:	3) Ch.2.3 Import in Unity Ch.2.4.1-2.4.3: Unity Animation Control in Mecanim
10. October:	4) Ch.2.4.4-2.4.5: Unity Mecanim Layers and Blend Trees
17. October:	5) **Milestone 1: Checkpoints 1-4 (30% Grade)**, Ch. 2.5: Setting up a Character Controller
24. October:	6) Ch. 2.6: Unity Blend Shapes
31. October:	7) Ch. 2.7: Unity IK
7. November:	8) Ch. 3: Navigation
14. November:	9) Ch. 4: Advanced Visual Effects
21. November:	10) Ch. 5: Performance Optimization
28. November:	11) **Milestone 2: Checkpoints 5-8 (30% of Grade)**, Project Definitions, Ch. 5: Performance Optimization
5. December:	**12) Adv. GameDev Project 1**
12. December:	**13) Adv. GameDev Project 2**
29. December:	**14) Adv. GameDev Project 3,** **Project Discussion, mandatory, Checkpoints**
9. January:	**15) Adv. GameDev Project 4**
16. January:	**16) Project Presentations Milestone 3: (40% of Grade)**

1.5 Tools

Beside Unity we will use other free tools that you can download at:

- **Unity**: https://unity3d.com/
- **Blender**: https://www.blender.org/
- **GIMP**: http://www.gimp.org/

Please bring always a mouse with you to the lecture. The touchpad of your laptop is not enough to fully operate Blender and Unity.

1.6 Additional Material

This **script** serves as protocol throughout the course and should help you to follow the tutorials in the course. It was created with great help from our students *Mark Wacker* and *Kaspar Schmid* and I am very thankful for their big effort.

The script is probably not perfect yet and we would appreciate any comments and suggestions for improvement on Google Docs. If you did not get shared this document for comments, please contact us.

We do not propose any **additional textbook** for this module because our topics are too heterogeneous. This doesn't mean that there aren't good books on advanced game development topic.

Because a protocol of our tutorials is sometimes too minimalistic, we provide many additional links to **video tutorials** or other resources that you can use in your self-study time. Please tell us if any of the links do not work anymore.

2 Rigged Skeleton Animation

2.1 Introduction

This part will focus on two main aspects of working with 3D characters. The first part will give an overview of the work required to create a new character model and bring it to life through animation. The second part will show how these characters and their animations can be used to achieve a believable and controllable avatar in a game engine. These tasks often span multiple fields and professions, from the concept artist that illustrates the initial look of the character to the technical artist that brings everything together through scripting.

2.2 3D Character Animation in Blender

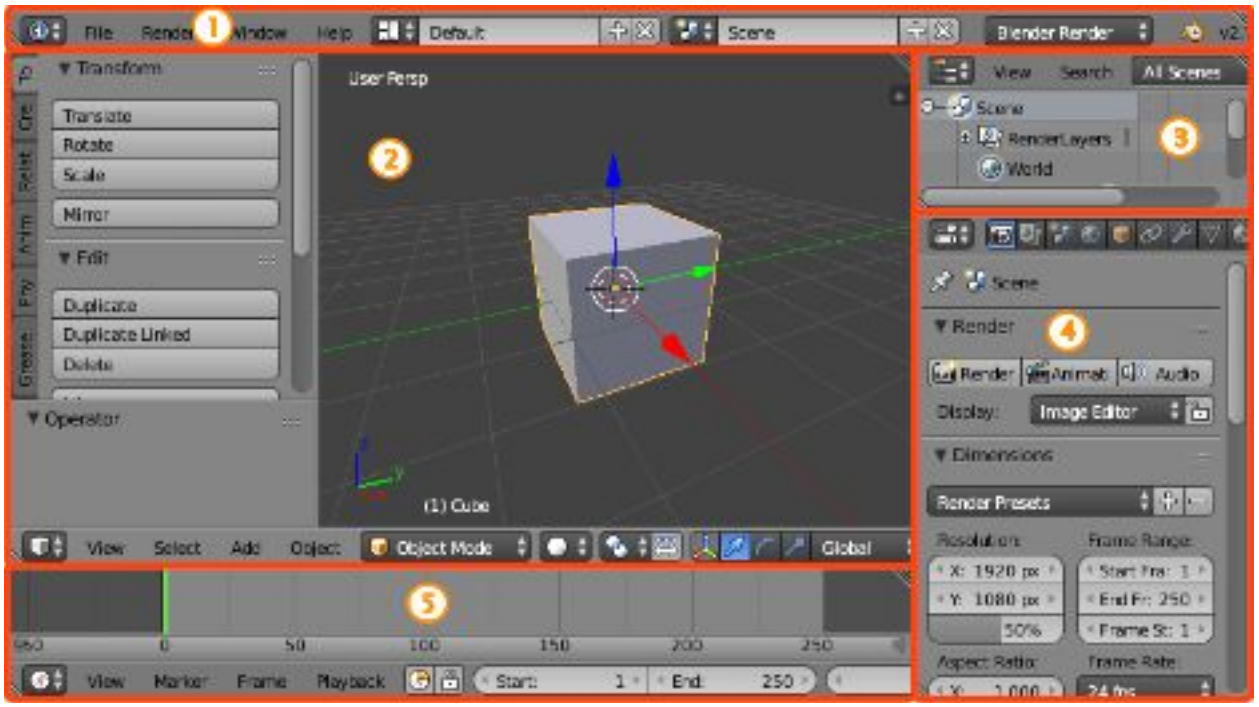
Creating a 3D character and its animations is no small feat, in bigger studios, this process often involves three or more specialized artists. A concept artist who comes up with the initial character design, a modelearer who translates the sketches and drawings into a full 3D model, a texture artist that creates the surfaces and material properties and finally an animator who breathes life into the character. Our focus here will be the tasks of the animator, however, all of these artists with the exception of the concept artist have a good understanding of how the others work.

For us as programmers, it is important to know how the artists work. Be it for knowing how to translate their output into a game, or for writing tools that help the artists work more efficiently or easier.

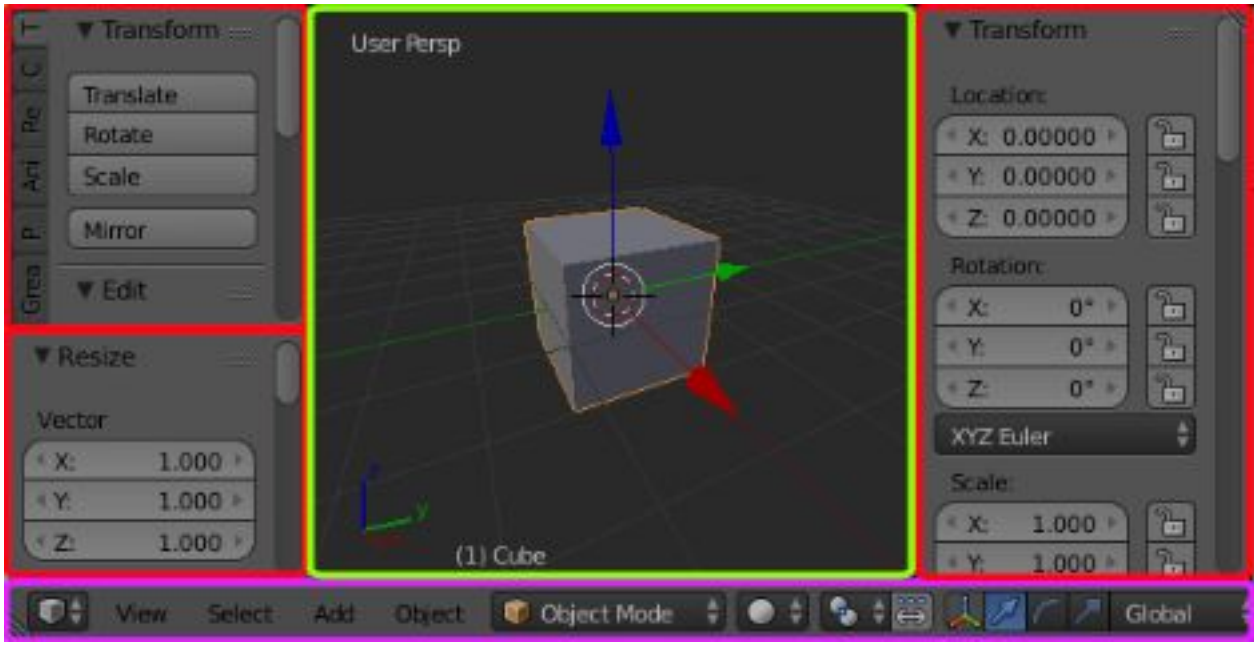
2.2.1 Blender overview

Blender is an open source software that has been initially released in 1995 and received improvements and new features ever since. You can download Blender at http://www.blender.org/.

Blender includes many features related to 3D modeling, it even has a video editor and a game engine built in. We're going to focus on the modeling and animation aspect of it however.



Blender's default Screen Layout with 5 editors: Info (1), 3D View (2), Outliner (3), Properties (4) and Timeline (5).



The 3D View as an example editor.

Regions

At least one region of an editor is always visible. It's called the main region and is the most prominent part of the editor. In the 3D View above this is marked with a green frame.

Aside from that, there can be more regions available. In the 3D View above these are the Tool Shelf (toggle visibility with T) on the left side and the Properties (toggle visibility with N) on the right side. They're marked with red frames. Additional regions mostly show context-sensitive content.

Each editor has a specific purpose, so the main region and the availability of additional regions are different between editors. See the specific documentation about each editor in the Editors chapter in the blender online manual.

See also the online documentation for a more detailed overview of the editor.

You can find additional information in the following video tutorials:

Darril Lile: Blender Quick Start Guide

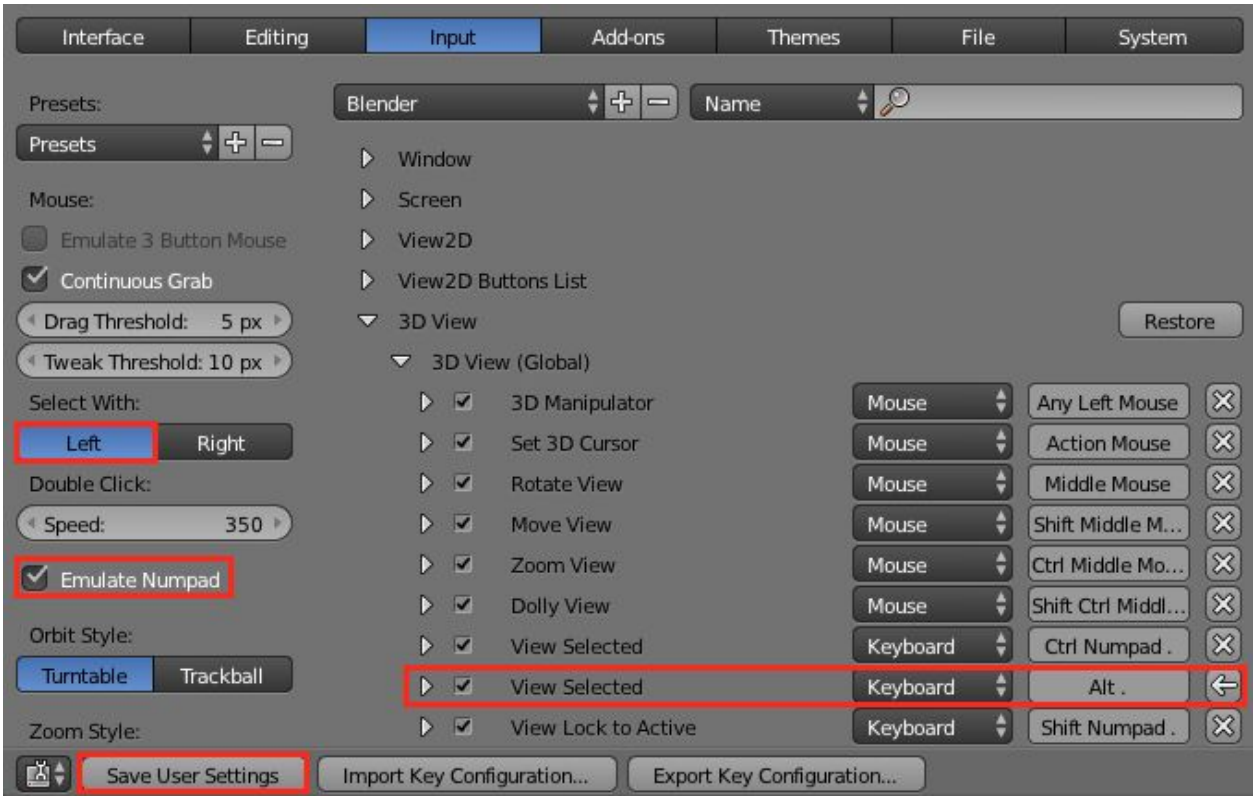
AgenZasBrothers: Blender for Beginners #1: User Interface (in German)

AgenZasBrothers: Blender for Beginners #2: Navigation (in German)

2.2.1.1 Setup laptop keyboard

If you don't have a numpad on your laptop keyboard you can install a python script that maps the number keys of the numpad to the main number keys of the keyboard:

1. Open the user preferences: *File > User Preferences > Input*
2. Choose the option *Select with Left*
3. Enable *Emulate Numpad*
4. Fold out the keyboard shortcut hierarchy to *3D View > 3D View (Global) > View Selected*: Click into the light gray text field and press [Alt-.]
5. Click Save on the bottom left of the preferences window



2.2.1.2 The most important shortcuts for a smooth start

Blenders hotkeys are relative to the mouse location, the window under the mouse dictates the function invoked by a key press.

Key	Functionality
G	Move (X, Y and Z to lock axis)
S	Scale (X, Y and Z to lock axis)
R	Rotate (X, Y and Z to lock axis)
Shift + C	Recenter 3D Cursor
N	Toggle properties region
T	Toggle tool shelf region
Tab	Switch between Edit Mode and previously active mode
A	Select/Deselect all
Z	Toggle between wireframe and solid display mode
Left Mouse Button (lmb)	Select object
Right Mouse Button (rmb)	Set 3D cursor in 3D View (context menu in other views)
MMB + drag	Rotate 3D View
Shift + MMB	Move 3D View
Mouse Wheel	Zoom
Numpad 0	Toggle Active Camera
Numpad-1/-3/-7	Front / Right / Top View
Numpad-2/-4/-6/-8	Orbit View Down / Left / Right / Up
Numpad 5	Toggle Orthogonal/Perspective view
Ctrl + .	Focus selected object(s)

More detailed shortcut lists:

http://www.katsbits.com/tutorials/blender/useful-keyboard-shortcuts.php

http://wiki.blender.org/index.php/Doc:2.4/Reference/Hotkeys/All

Or check *File* > *User Preferences* inside Blender, or use the context menu in Blender (space) to find shortcuts.

2.2.2 Doing all low level

Character animation is one of the most complex parts in real-time computer graphics. Even today with dozens of 3D data formats there is no common denominator on how to define a bone model (the rig). We will therefore first have a look on the hard way, doing all bottom up in Blender.

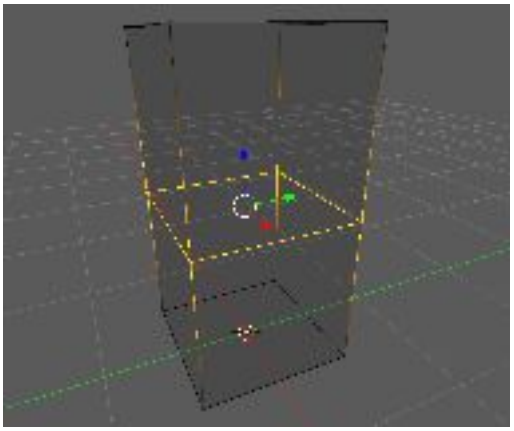
The following tutorial is mostly based on the video tutorials on Blender Character Creation & Animation by Sebastian Lague. His tutorials have a good mix of simplicity and completeness.

There is another complete tutorial on character modelling, rigging and even Unity import by Darrin Lile. But this tutorial is far more complex and covers many artistic details.

2.2.2.1 Creating a simple animated box in Blender

The goal of this tutorial is to create the simplest animation for a bending box. We will first create a box, add some vertices in the middle and then add two bones that build a so-called rig. The rig is a virtual skeleton of a 3D object. The purpose of the rig is that we can deform the surrounding object only be moving the rigs bones.

1. **Creating a Box:**
    a. New project [ctrl-N]
    b. Select camera and lamp and delete them with [X]
    c. Move cube up by 1 unit to change the pivot to be at its bottom center [G,Z,1,enter]
    d. Scale the cube
        i. to half its size on the X-axis (to get an elongated box) [S,X,0.5,enter]
        ii. to half its size on the Y-axis [S,Y,0.5,enter]
    e. Apply the current transformation (location and scale) (normalizes scale factor and resets position, this will finally move our pivot point) [ctrl-A]. You can check the absolute dimensions in the properties region [N]. All scale factors are now 1.0.
    f. Switch into *Edit Mode* [tab] and view the cube in wireframe [Z]
    g. Add a horizontal loop cut to the box's center [ctrl-R + enter + enter]



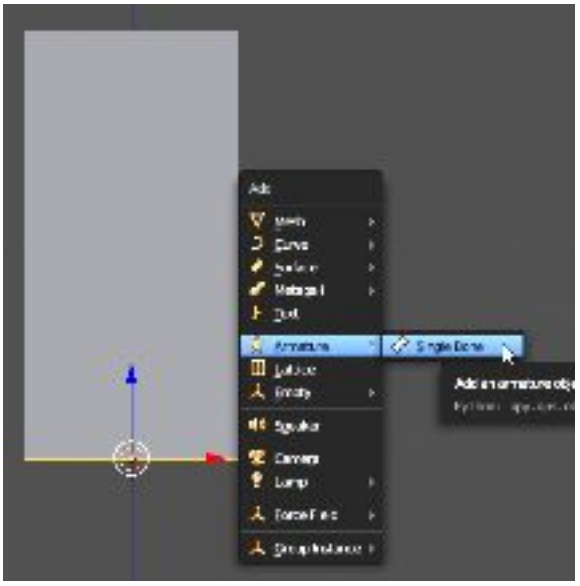
2. **Adding two Bones:**
    a. Switch into orthogonal view [5] (alt: bottom left '*view* > *view persp/ortho*')

b. Switch into front view [1] (alt: bottom left *View* > *Front*)

c. Make sure the 3D cursor is at 0 [shift-C]

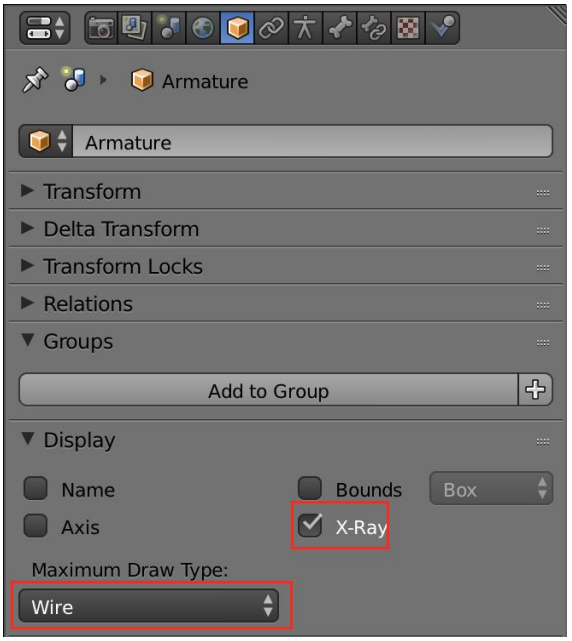
d. Switch back into *Object Mode* [tab]

e. Add an armature to the scene by selecting *Armature* > *Single Bone* [shift-A]



f. With the armature selected, go to the properties panel and switch to the *Object* tab (orange cube icon).

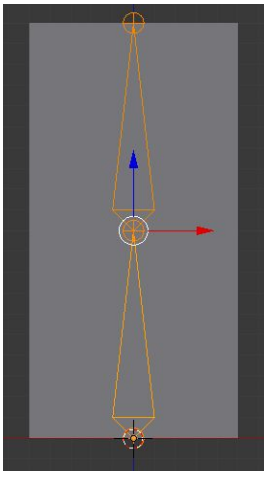
g. Under display check the *X-Ray* and Names box and switch the *Maximum Draw Type* to *Wire*.



h. Switch into *Edit Mode* [tab]

i. Select the tip of the bone (top circle) with lmb

j. Press [E,Z,1,enter] to extrude a new bone along the z-axis with length 1.

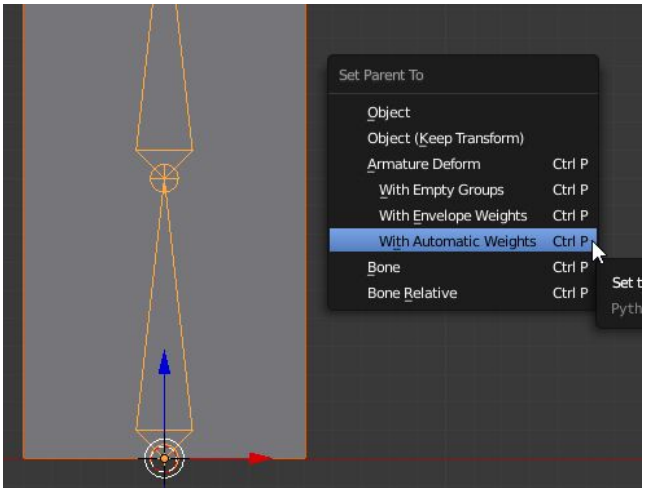


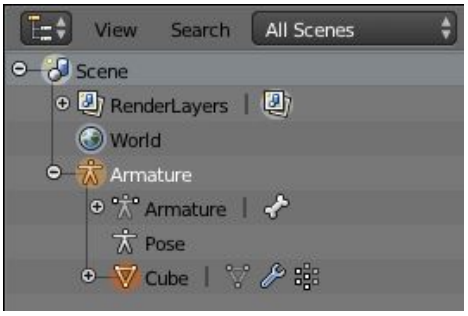
3. Parenting the Mesh to the Armature:

a. Back into *object mode* [tab].

b. Select first the cube and then Shift + select the armature

c. Parent the armature to the cube by selecting *Armature Deform > With Automatic Weights* after [ctrl-P]. The cube becomes a child of the armature as can be seen in the outliner editor that displays the scene graph structure.



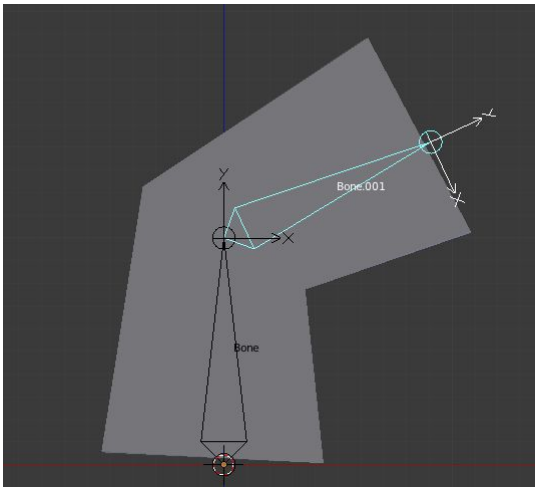


4. Switch to Pose Mode:

a. Select the armature and switch into the *Pose Mode*.

b. Switch to front view [1] and select the top bone and rotate it with [R] and the LMB: The bottom vertices bend with the top bone. This is because of a non-zero vertex

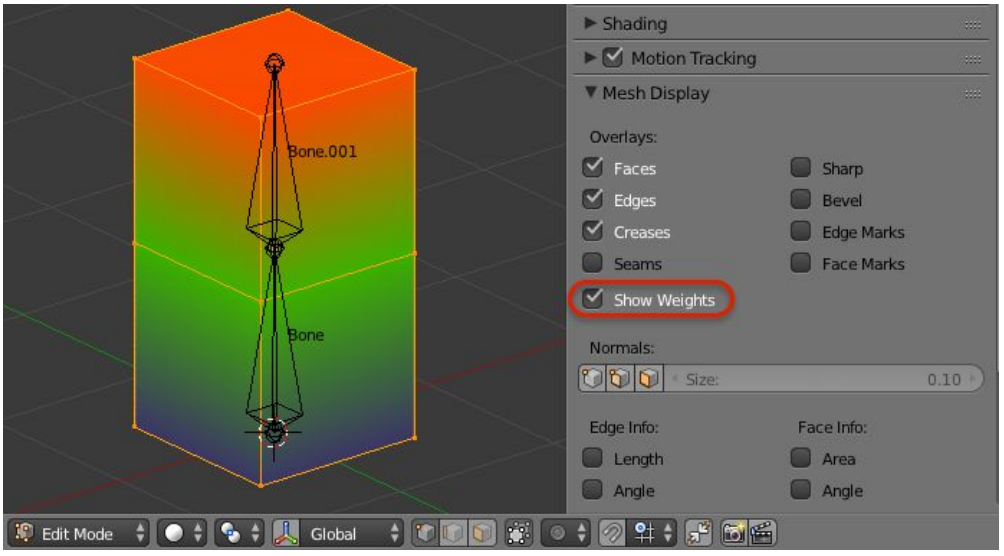
weight for the top bone the bottom vertices.



5. Show the Vertex Weight Colors:

a. In *Object Mode* select the cube and go into *Edit Mode* [tab].

b. In the properties panel (N to toggle) turn on the *Show Weights* option. The red color shows a vertex weight of 1 and the blue color is vertex weight of 0.

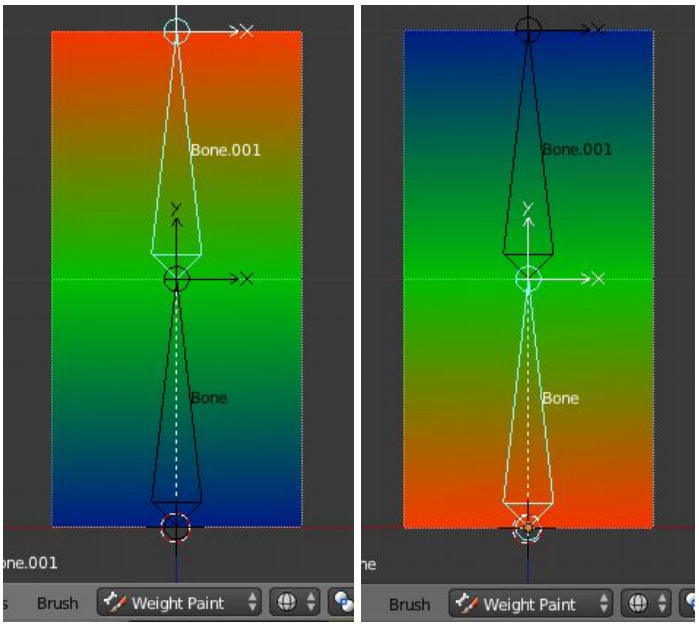


6. Vertex Weight Painting: We can modify now the vertex weight with *Weight Painting*.

a. In *Object Mode* select the armature and switch to *Pose Mode*. Select the top bone.

b. Switch to front view [1] and wireframe view [Z].

c. In *Pose Mode* select the mesh and then switch into *Weight Painting* mode. You can now select the bones and see how the color changes for the bones:

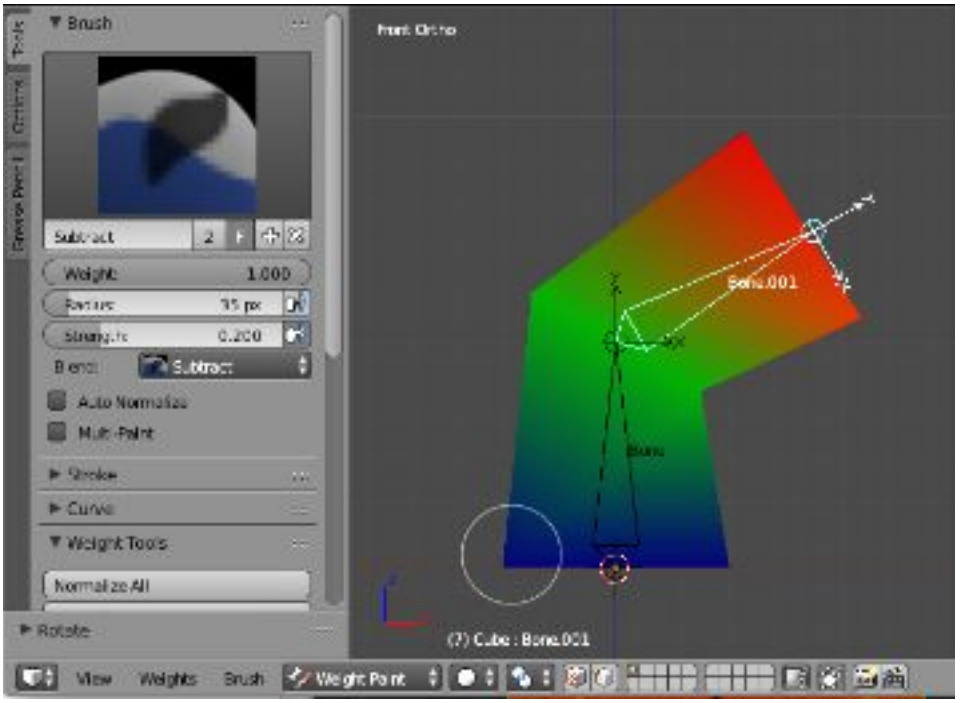


d. In *Weight Paint* mode we can define a brush to apply weights to an area under the cursor.

i. Open the *Tools* panel [T] and choose the *Add* brush and paint over the reddish vertices to make them pure red.

ii. Choose the *Subtract* brush and make the blueish vertices pure blue.

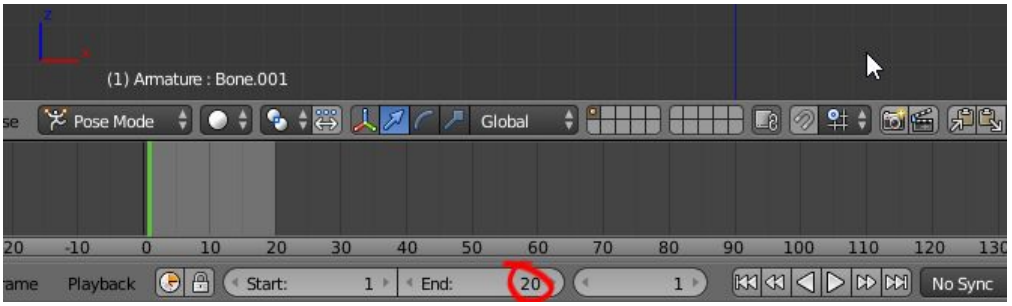
iii. Do this for both bones and be sure that you are in wireframe mode [Z]



e. Select any of the bones and rotate it a bit [R] to see the effect of the vertex weights. If you right click while rotating the rotation will reset and not be applied.

7. Adding a Simple Animation:

a. In the *Timeline* panel set the end frame of the animation to 20

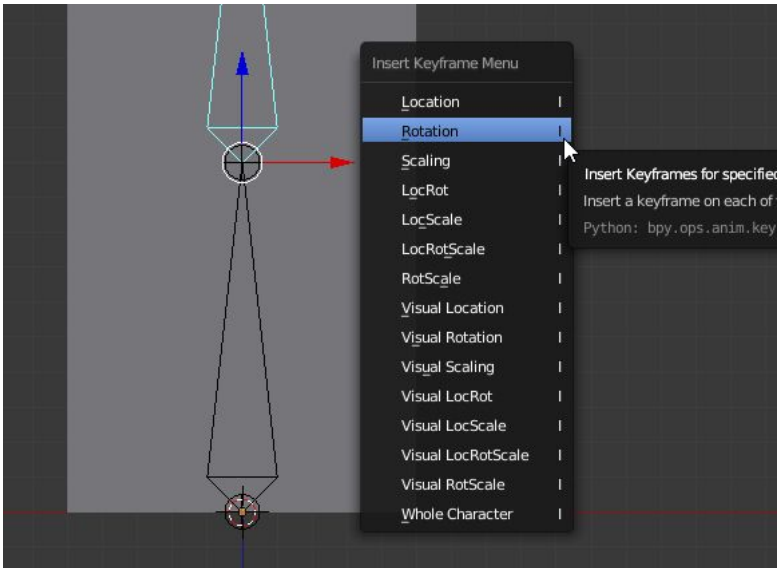


b. Zoom the gray timeline area with the mousewheel. Move it with shift-mousewheel.

c. In the *Pose Mode* select the bone you want to animate

d. Position the scrubber (the green line) in the timeline to position 0.

e. Insert a keyframe [I] for the rotation.

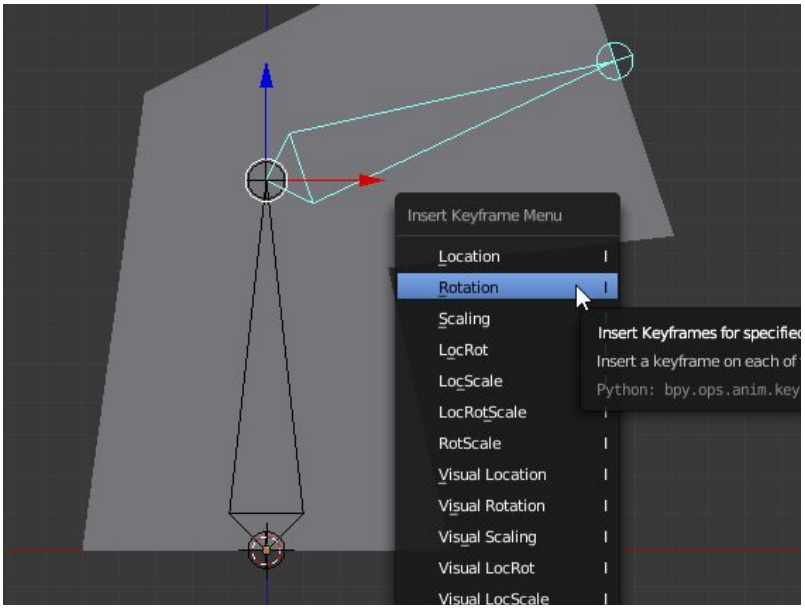


f. Do the same at the end of the animation (frame 20)

g. Move the scrubber (the green line) with rmb on the timeline to the center

h. Now rotate the bone around a bit [R]

i. Add a keyframe for the new rotation [I]



j. You should now have a simple animation of your cube twisting or bending. You can play back or forward the animation in the timeline. 

k. Save your animated box with [ctrl-s] under any name and in a folder named *02.2.2.Do it all low-level*

Checkpoint 1: Animated Box

2.2.2.2 Blender Character Creation: Modelling

This tutorial follows the 1st part of Sebastian Lague's Youtube tutorial.

Because this tutorial is about pure modeling in Blender, we won't go through it in this course. You learn a lot about the basics of Blender and in special about creating a character model.

2.2.2.3 Blender Character Creation: Texturing

This tutorial follows the 2nd part of Sebastian Lague's Youtube tutorial.

Because this tutorial is about pure modeling in Blender, we won't go through it in this course. You learn how to texture map the previously created character model.

2.2.2.4 Blender Character Creation: Rigging

This tutorial follows the 3rd and 4th part of Sebastian Lague's Youtube tutorial.

Bones are added to the model and create the so-called rig that is nothing else than the skeleton of the character. The purpose of the rig is that we can deform the character mesh as easy as possible for creating nice animations later on.

2.2.2.5 Blender Character Animation: Walk Cycle

This tutorial follows the 5th part of Sebastian Lague's Youtube tutorial.

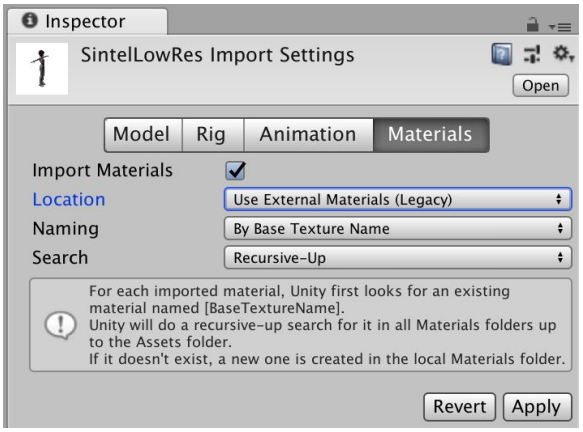
For this part, there won't be a step by step list since it is more of an artistic process. Sebastian Lague shows how to create a very nice walk cycle animation.

2.2.3 High-level Rigging with Blender Rigify

In this tutorial, we will rig the Sintel character from the Blender Movie Sintel with the Blender Addon Rigify. The Sintel character can be downloaded from various sites and used freely with the creative commons license. Our model is a lowres version and adapted from the Sintel Game repository. A highres version can be found at BlendSwap.

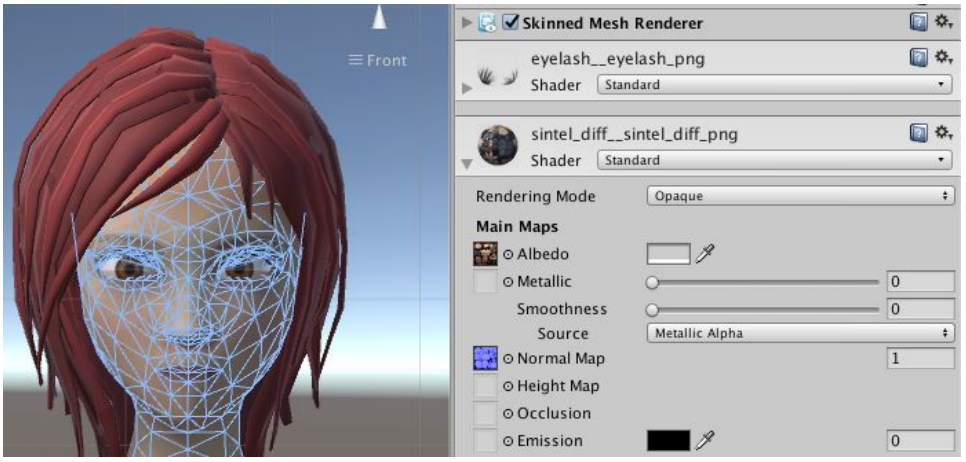
2.2.3.1 Import a Blender File in Unity

1. **Import Sintel into Unity**:
    a. Create a new Unity 3D project and name it *02.2 High-level Rigging for Sintel*.
    b. Copy the *SintelLowRes.blend* model file with the Textures folder from the Dropbox into a new subfolder *Assets/Models/Sintel*. Unity will convert the Blender file in the background with the Blender FBX converter.
    c. To make the materials editable select the imported model, go to the material tab of the import settings and choose *Use External Materials (Legacy)* and click *Apply*:



    d. Create a plane at [0,0,0].
    e. Place the *SintelLowRes* model at [0,0,0]
2. **Setup the Materials**: Materials from within Blender files have often the wrong assignment.
    a. Create 4 materials and assign them to the different parts of Sintel:
        i. *Sintel_eyeball_diff* for objects *sintel_eye_L* & *sintel_eye_R*:
        - Assign the Sintel_eyeball_diff.png to the *Albedo* field.
        - Set the shader *Standard (Specular setups)*
        - Increase the *Smoothness* to 0.9.
        ii. *Eyelash* for object *sintel_body*:
        - Assign the *eyelash.png* texture to the *Albedo* field.
        - Set the Rendering Model to *Fade*.
        iii. *Sintel_diff* for object *sintel_body*:
        - Assign the *sintel_diff.png* texture to the *Albedo* field.
        - Assign the *sintel_nrm.png* texture to the *Normal Map* field.
        - Reduce the *Smoothness* to 0.1.

iv. *Sintel_hair_solid* for object *sintel_hair_soild*:
- Assign the *sintel_hair_solid.png* texture to the *Albedo* field.
- Adjust the *Smoothness*.

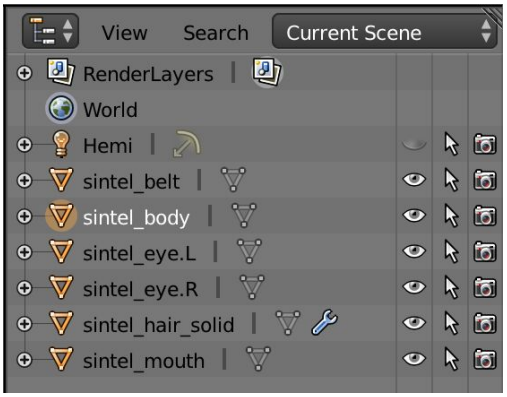


b. Save the scene with CTRL-S

2.2.3.2 Setup Rigify Rig in Blender

Rigify is an add-on to Blender that generates us a professional and highly customized rig with hundreds of control bones with full IK (*Inverse Kinematic*) control over legs, arms and fingers. First, we will add and adjust a so-called metarig. After the correct in body placement of the metarig you can let the final Rigify rig be generated automatically.

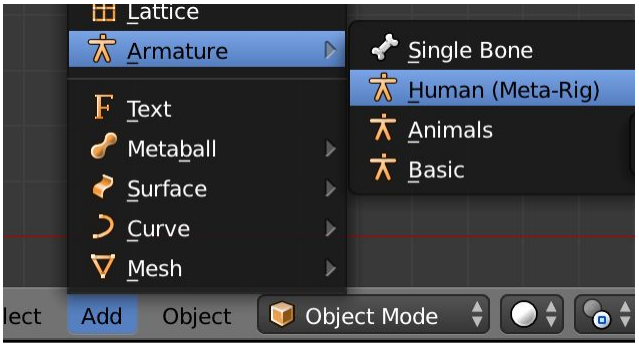
1. **Open Blender** by double-clicking the *SintelLowRes* file within Unity.
   a. **Enable the Rigify Addon:**
      i. Select *File* > *User Preferences …* [Cmd+,]
      ii. Go to the *Add-ons* tab: Search for the *Rigify* add-on and mark it.
      iii. Go to the *File* tab: Mark the *Auto Run Python Scripts* option.
      iv. Click at the bottom *Save User Settings*
   c. **Preparations**: The Sintel has 6 objects with meshes:



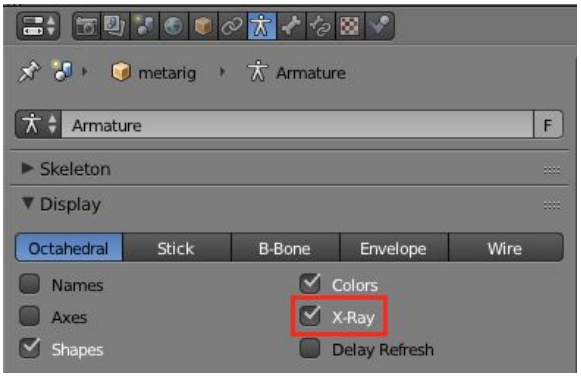
Before we add the Rigify rig, we have to **be sure that**:

i. The model stands on the ground at [0,0,0].
ii. All objects are not transformed. You can see that by opening the side panel with N. The *Location* and *Rotation* values must be 0 and the *Scale* values must be 1:
iii. If one object has a transform you can apply it with [ctrl-A].

3. **Add and adjust the Metarig**:
   a. Add a Rigify metarig with:



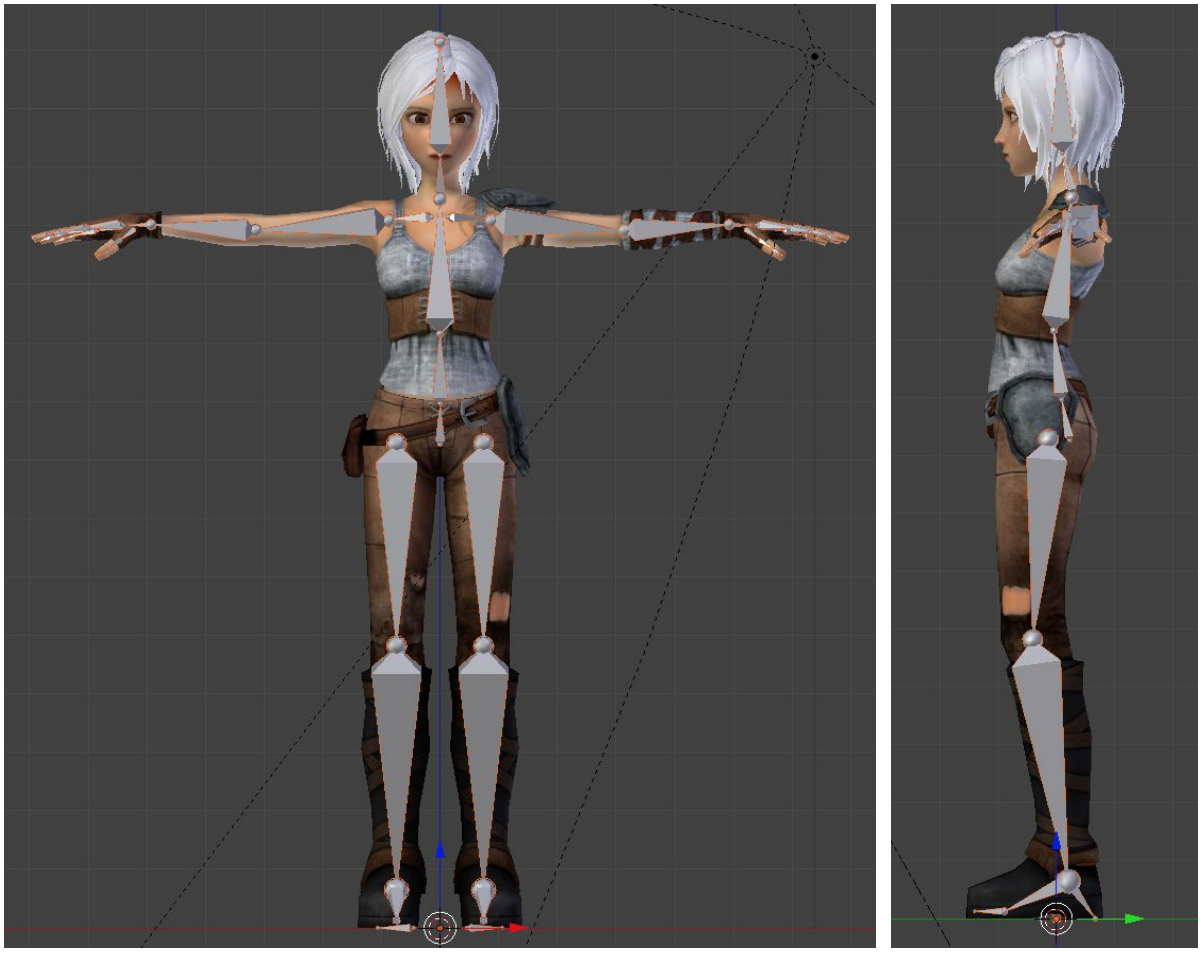
   b. View the model from the orthographic front [1].
   c. Make the rig to be shown always on top with the option X-Ray Armature property panel:



   d. Select the metarig and reset it's *Location* to [0,0,0] in the side panel.
   e. **Scale the metarig** in *C* with S to the size of the Sintel mesh.
   The top bone must be at the top of the head.
   f. Apply the scale with [ctrl-A > Scale].
   g. Go into *Edit Mode* with [tab].
   Important: Before we adjust the single bones make sure to mark the ***X-Axis Mirror*** option in the Options tab of the left side panel [T]. All symmetric bones will be edited symmetrically.

h. **Adjust the single bones** of the metarig:
- Swap between the front [1], side [3] and top view [7].
- Swap between the *Solid* and *Wireframe* shading mode with [Z].

i. Select one or more bones or joints that you want to place anatomically correct and move with [G] or rotate with [R].



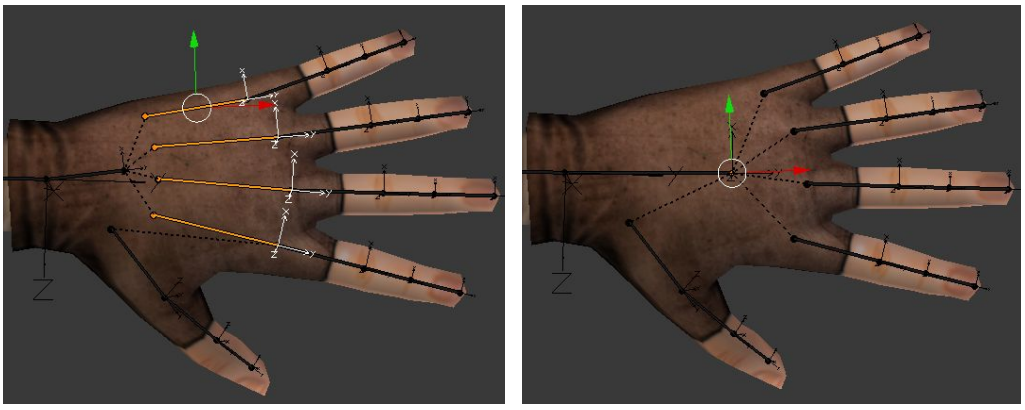
There now two possibilities to continue with Rigify described in the next two chapters:

1. **Continue with the Rigify Metarig:**
   a. **Advantages**: The metarig is fairly simple without a complex IK setup. The import into Unity is straightforward and Unity will recognize the metarig after the removal of some additional bones.
   b. **Disadvantages**: We will have to pose our animations completely with FK. Or we add our own IK constraints.
2. **Generate the Rigify Control Rig**:
   a. **Advantages**: You will get a professional IK rig for easy posing.
   b. **Disadvantages**: The generated rig is complex and needs to be adapted so that we can import it into Unity. There exist multiple recommendations and even a special Unity Asset for these adaptations. None of them is really easy.

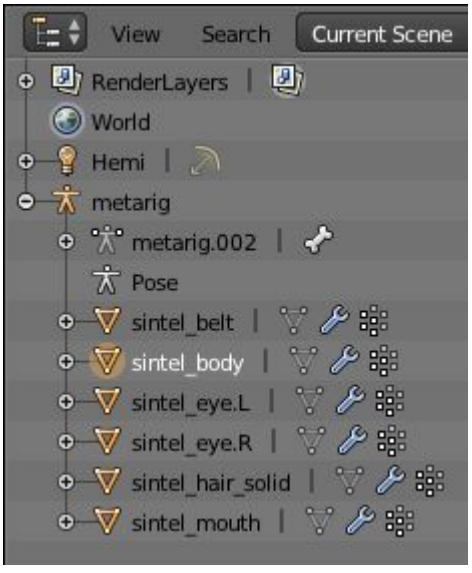
2.2.3.3 Continue with the Rigify Metarig

If we continue with the Rigify metarig we have to remove some bones so that Unity later can recognize the rig as a humanoid rig.

1. **Delete some bones**: Over the years the Blender Rigify rig got more and more detail bones. We do not need all of them. In the *Edit Mode* **delete** the following ones:
    a. The *breast* bones
    b. All *face* and *ear* bones. You can block select with [B] or cursor select with [C].
    c. The bone named *face* within the *spine.06* bone
    d. The *heel* bones
    e. The *pelvis* bones
    f. The *palm* bones. Move the endpoint of the hand bone closer to the fingers:



2. **Make the mesh objects a child from the rig**: Before we can move the mesh by the rig we have to make it a child from it.
In *Object Mode* select all mesh nodes, then hold shift and select the rig, press [Ctrl+P] and select *Armature deform > With Automatic weights*. This will assign the mesh to the bones creating vertex groups, this is what we will use for the skinning process. After that, all meshes became a child of the *metarig* and got an *armature* modifier:

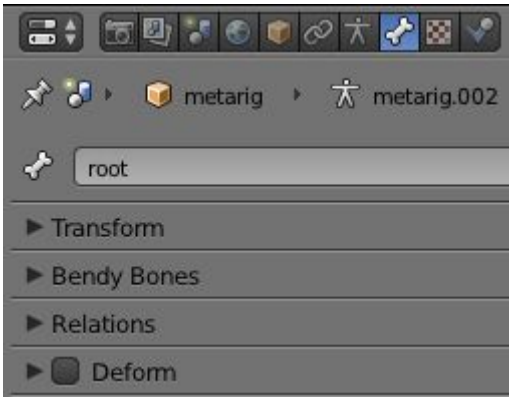


3. **Check the Pose Mode**: If everything went ok you should be able to go into the *Pose Mode* and if you rotate a bone the mesh should be deformed as well.

4. **Adding a Root Bone**: A root bone allows to move the complete rig together:

    a. In *Edit Mode* press [shift-C] to center the 3D-cursor.

    b. Press [shift-A] to add a new node and rotate it with [R,X,-90]. Scale it to the front with [S,Y] and name the bone *root* and deselect *Deform* in the *Bone* tab of the property window.

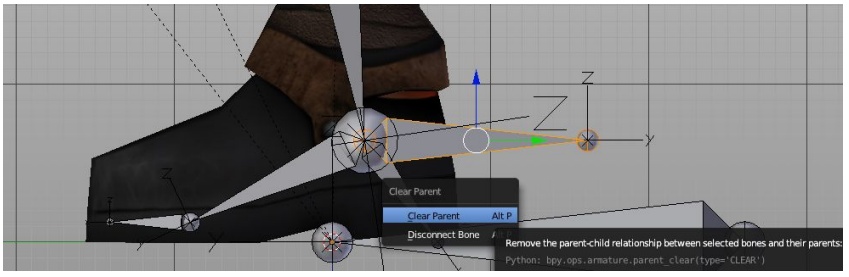


    c. Now parent the *spine* bone to the *root* bone by selecting the *spine* bone first and then shift-select the *root* bone and press [Ctrl-P] and choose *Keep Offset*.

5. **Add Inverse Kinematik Control Bone for the Leg**:
So far we can only control the rig with the so-called ***forward kinematics (FK)***. To be able to control the rig easier for animation we add some control bones with *inverse kinematic* constraints. This is just the hierarchical control of the parented bones. If we move the upper leg, all lower bones will follow. To design animations only with FK is very difficult. We mostly want to move a foot or a hand to the right place and all the upper bones should rotate and move naturally. That is what ***inverse kinematics (IK)*** does. But our knee or elbow cannot bend freely. That's where ***IK constraints*** play an important role.

    a. In *Edit Mode* select the lower joint of the *shin.L* bone and extrude a new bone with [E,y] along the y-axis. If X-Axis Mirror is still set you should you got two new bones. Name them *Leg_IK.L* and *Leg_IK.R*.

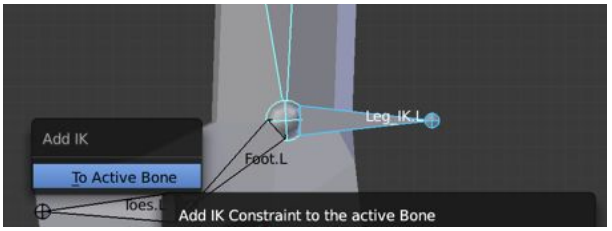
    b. Select the new *Leg_IK.L* bone and press [alt-P] and choose *Clear Parent*.



    c. Turn off the *Deform* property in the *Bone* property tab because these bones are not control-bones and will not deform the mesh.

    d. In *Pose Mode* select the control bone *Leg_IK.L* and shift-select the *shin.L* bone and press [shift-I] and choose *Add IK To Active Bone*. The lower leg bone was the active bone and gets now yellow to visualize that it has now an IK

constraint.



e. In the *Bone Constraints,* tab set the *Chain Length* to 2 so that the lower and upper leg gets moved by the *Leg_IK.L* bone.

f. In *Pose Mode* move [G] around the *Leg_IK.L* bone and watch how the leg bones get moved. If you move too much in front the leg suddenly bends backward. To avoid this backward bend we have to add a so-called *Pole Target*.

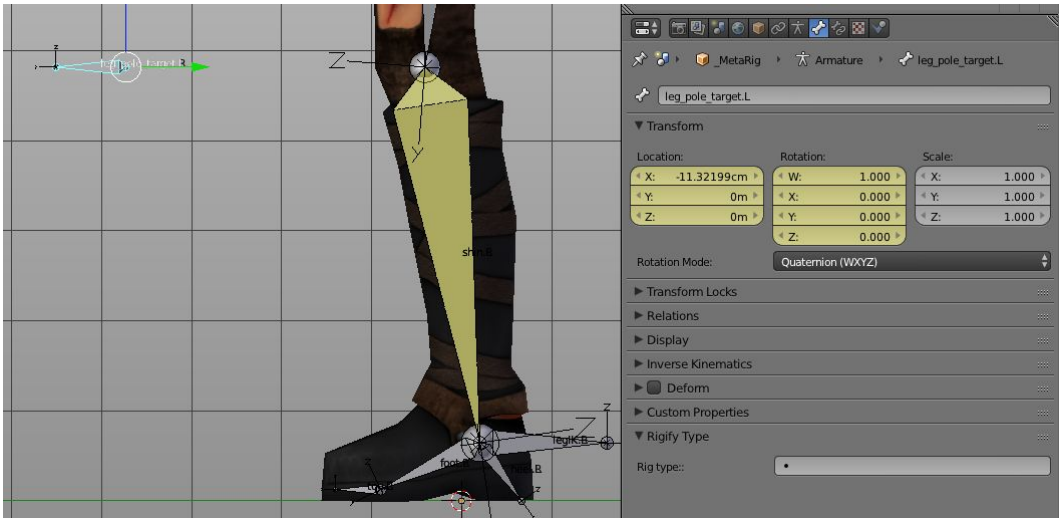
g. In *Edit Mode* select the upper joint of the *shin.L* bone and

   i. Extrude a new bone to the front [E,Y].

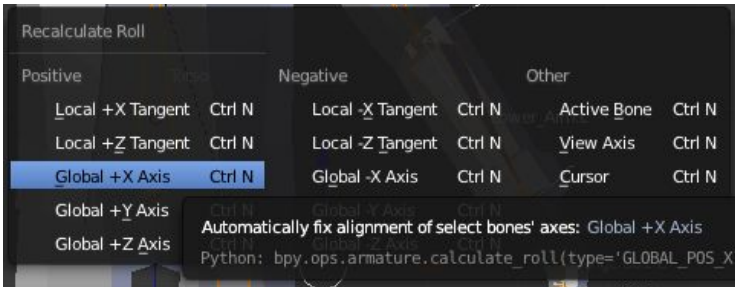
   ii. Select the new bone and clear the parent with [alt-P].

   iii. Move the bone to the front with [G,Y] so that the knee can't touch it.

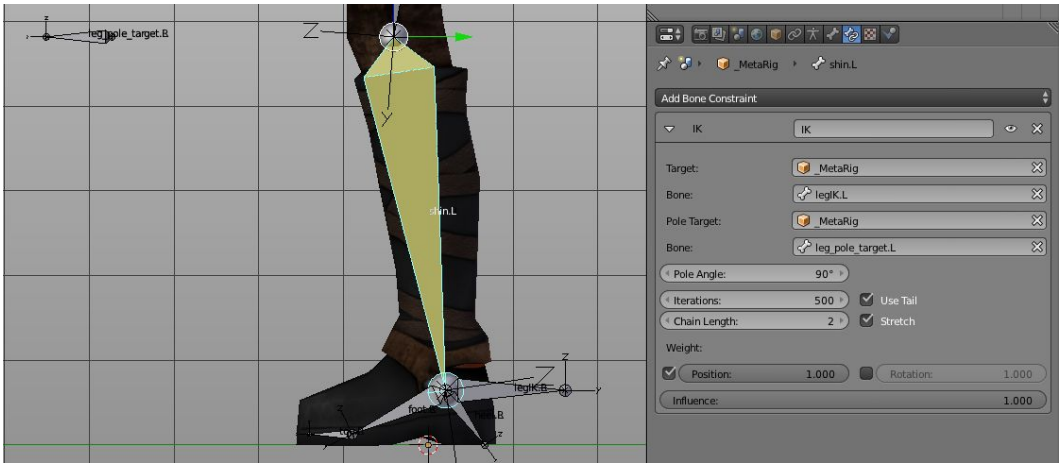
h. Rename the new bone to *Leg_Pole_Target.L* and unclick the *Deform*:



i. Before we add the pole target to the bone constraint we recalculate the bones roll axis. Select all bones with [A] and then press [ctrl-N, Global +X Axis], [ctrl-N, Global +Y Axis] and [ctrl-N, Global +Z Axis]. After this, all bone joints should have their primary roll axis around the x-axis.



j. In *Pose Mode* in the *Bone Constraints* tab of the *shin.L* bone set the following fields:
- Pole Target: *metarig*
- Bone: *Leg_Pole_Target.L*
- Pole Angle: 90°



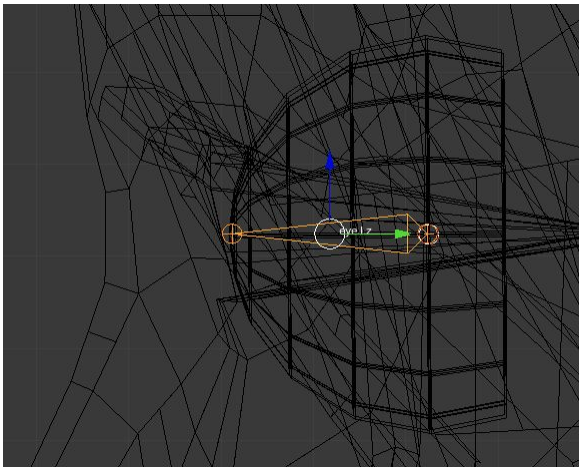
Now the leg should not anymore bend backward.

k. Repeat the same to add the *IK constraint* to the right leg.

6. **Adding Eye Bones**: Unity also allows eye bones in the standard humanoid rig. With these, you can let look the character in a specific direction. This is very important to give a character a human touch. The following steps are taken from the video tutorial on [eye rigging]:

a. Select the rig and switch to edit mode.

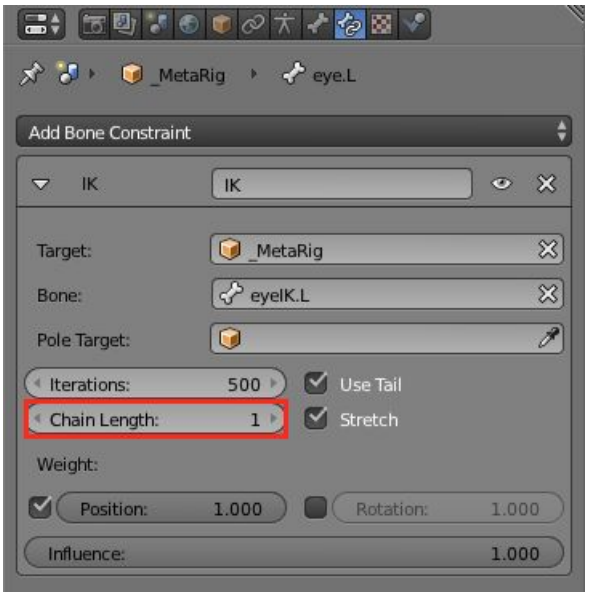
b. **New eye bone**: Create a new bone with [shit-A] and name it *eye.L* and place it into the center of the left eye. For better placement, you can hide everything except the eyes and use the wireframe display mode [Z].



c. **Parent eye to head**: Make the eye bone a child from the head bone by selecting first the eye bone and then shift-select the *3* bone and then press [Ctrl-P] with keep offset.

d. **Target IK bone**: Go to the side view with [3] and duplicate the eye bone with [shift-D] and shift it to the left on the blue y-axis. Name the new bone to *eye_IK-L*.and unckeck *Deform*. Clear the parent with [alt-P].

e. **Add IK-constraint**: To let the eye bone look always towards the new IK bone we add a constraint to it. In the pose mode select the *eye_IK.L* bone and shift select the eye.L bone and press [Shift-I] to add the constraint. In the bone constraint tab choose to set the *Chain Length* to 1:

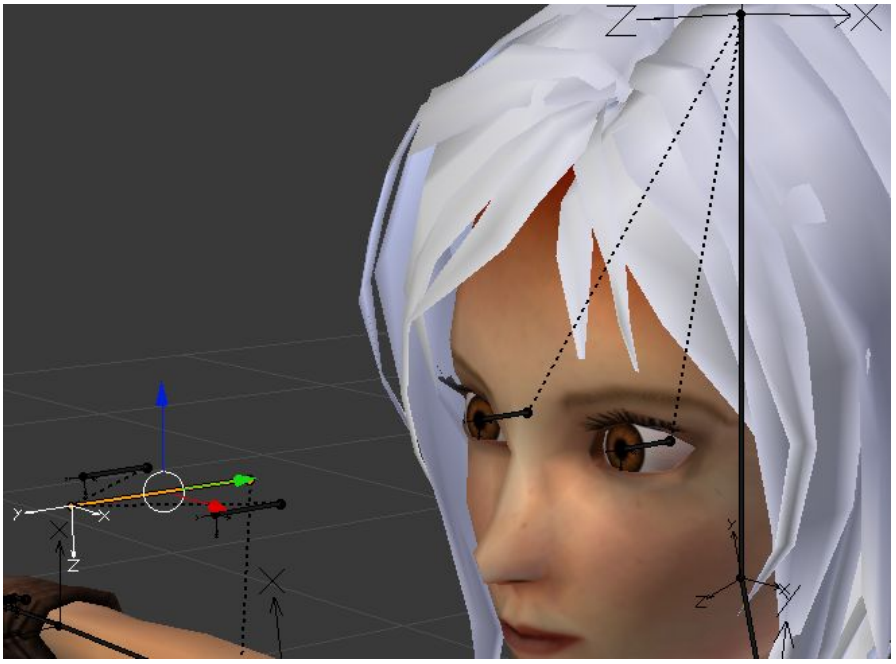


f. **Make the eye a child of the eye bone**. The easiest way to let the eye follow the eye bone we make the eye mesh object a child from the eye bone. Select first the left eye and then the *eye.L* bone and press [Ctrl+P] with *Armature deform > With Automatic weights*. Leave the *Deform* option in the bone tab of the eye bone checked, because only deform bones will be exported to Unity.

g. **Duplicate everything for the right eye** with [Shift-D]. Rename the copied bones *eye.R* and *eye_IK.R*. Parent the right eye to the *eye.R* bone.

h. **Single eye IK control bone**: Because both eyes look always in the same direction we add another bone in the center of the eyeIK bones and parent it

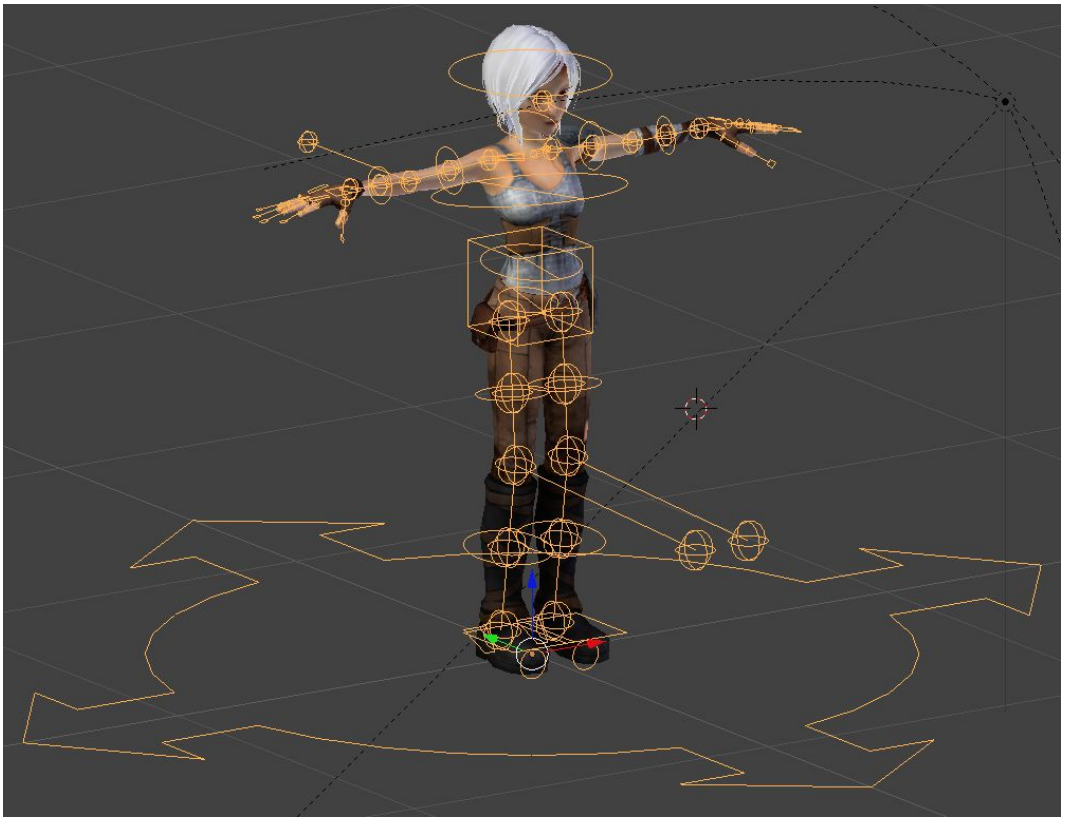
to them with [Ctrl-P].



2.2.3.4 Generate the Rigify Control Rig (2016)

1. **Generate Rigify Rig:**
    i. Go into Object mode and select the metarig.
    j. Select the Armature property tab and click the button *Generate* in the section *Rigify Buttons*.
    k. Check that no error messages appeared in Blender's menu bar.
    l. Hide the *_MetaRig* object with the eye symbol button. In the end, you could delete the metarig because we don't use it anymore.

m. Select the new *rig* object and choose the *X-Ray* option in the *Display* properties:

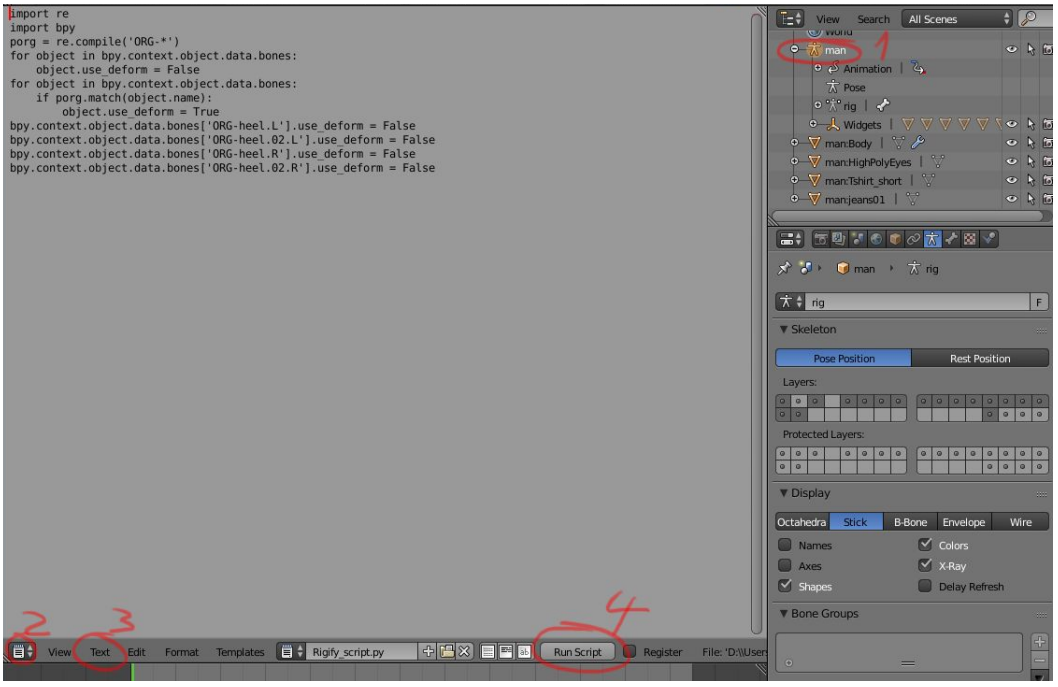


n. All the orange objects are control-bones that we can show or hide with the layer buttons. They are visible as small buttons with dots in the *Layers* section of the *Armature* tab or in the *Rig Layers* section of the right side panel.

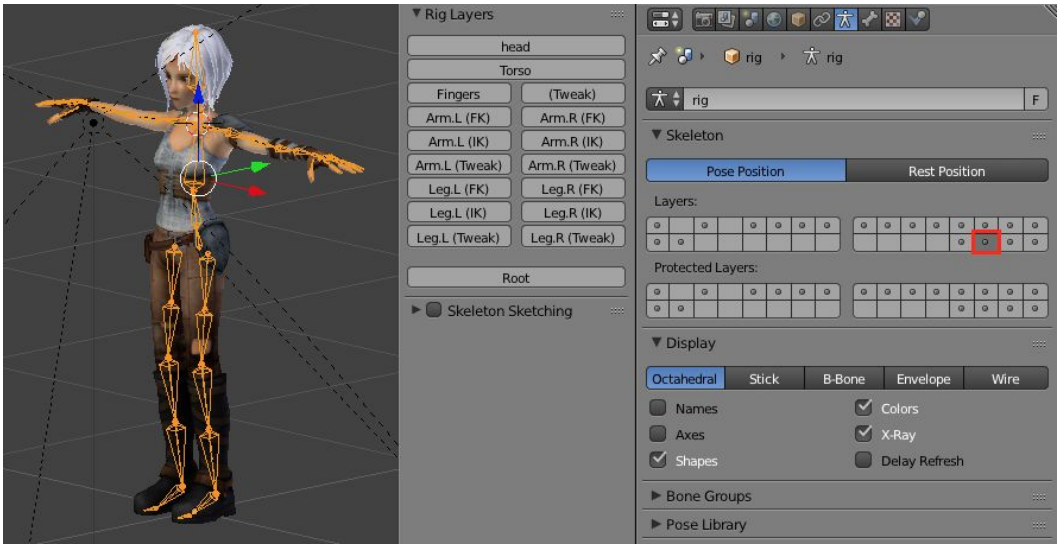
4. **Cleanup for Unity**: For our rig to work in Unity later we need to make a few adjustments.

   a. **Delete all WGT-Objects**: You will notice in the top right corner in the scene's hierarchy you have many WGT-bones (WGT stands for Widget). As these bones will not be skinned to the mesh or used at all, we will need to delete them to avoid having unassigned meshes in our Unity project. You can select multiple with box-select [B] and delete them in the context menu > delete.

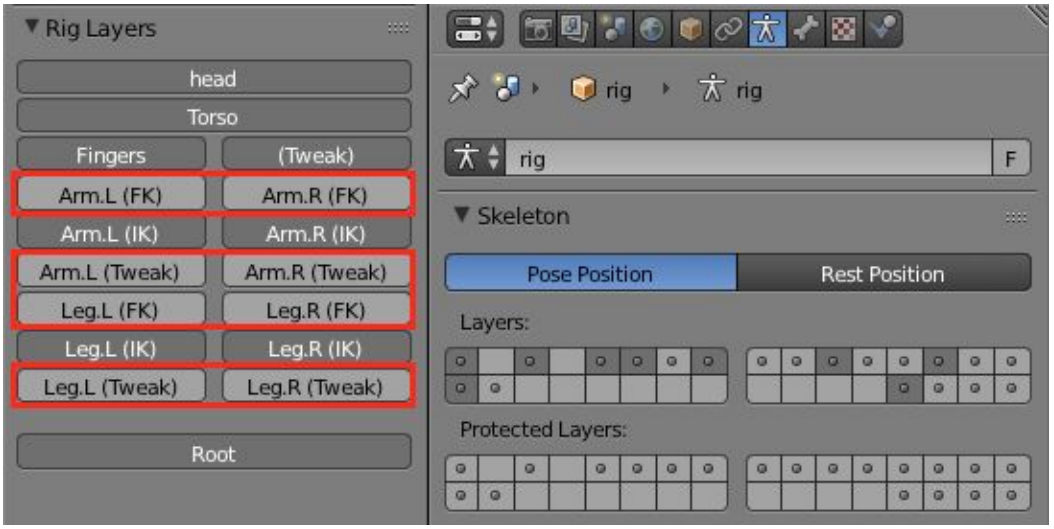
   b. **Rigify Script**: Select the armature and change the 3D View to a Text Editor. Go to *Text* > *Open Text Block* [Alt+O] and open the provided Blender script *Rigify_script.py*. Click the *Run Script* button and switch the window back to a 3D View. The script sets all bones with ORG-prefix to deformable so that they will be imported by Unity.



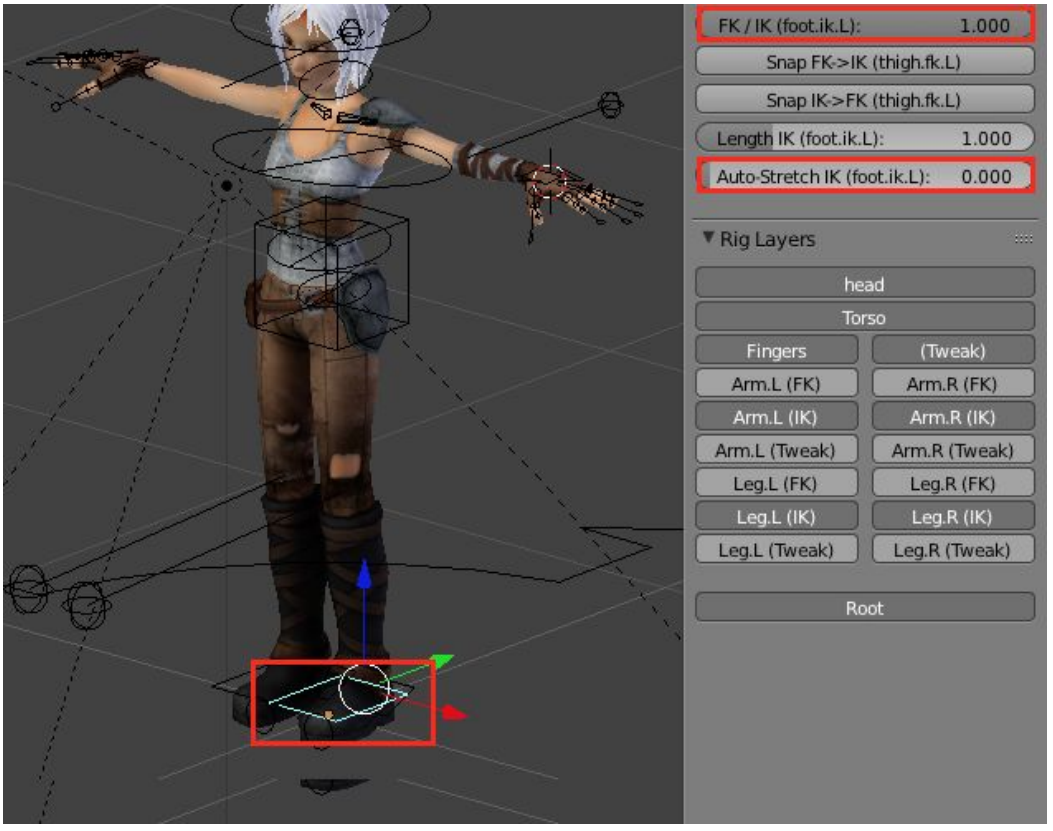
c. **Delete the unused DEF-* bones**: Select the armature and only show the deformation layer as seen in the next screenshot.
Switch into *Edit Mode* and select all bones [A] and delete all the bones on this layer [X].



5. **Make the mesh objects a child from the rig**: Before we can move the mesh by the rig we have to make it a child from it.
In *Object* mode select all mesh nodes, then hold shift and select the rig, press [Ctrl+P] and select *Armature deform > With Automatic weights*. This will assign the mesh to the bones creating vertex groups, this is what we will use for the skinning process.
6. **Hide FK-Bones**: We don't need the control bones for the forward kinematics (FK) of the arms and legs and we don't need the tweaking bones. We can deselect them best on the right side panel:



7. **In Pose Mode set 100% IK on Arm- and Leg-IK-Bones**: To be able to move the legs and arms with inverse kinematics (IK) we have to pull for all these control bones the slider *FK/IK (...)* in the *Rig Main Properties* section of the side panel to 1.0. In addition you can set the *Auto-Stretch IK* to 0.0.
   The next screenshot shows this for the left foot IK-bone:



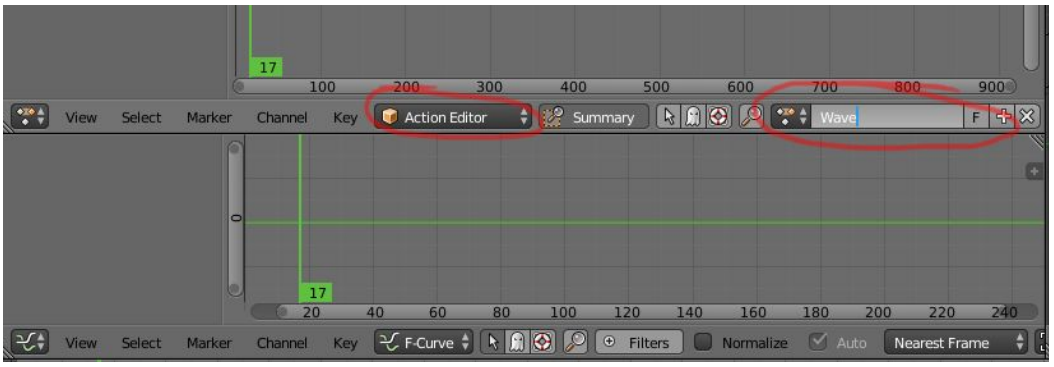
2.2.3.5 Create a simple Wave Animation

8. **Animation Setup**: In the menu bar on the right side of the *Help* button there are editor layout presets, select the *Animation* preset. Your editor is now split horizontally, on the left side you find the *Dope Sheet* above and the *Timeline* editor below.

a. In the *Dope Sheet,* window menu switch to the *Action Editor*.

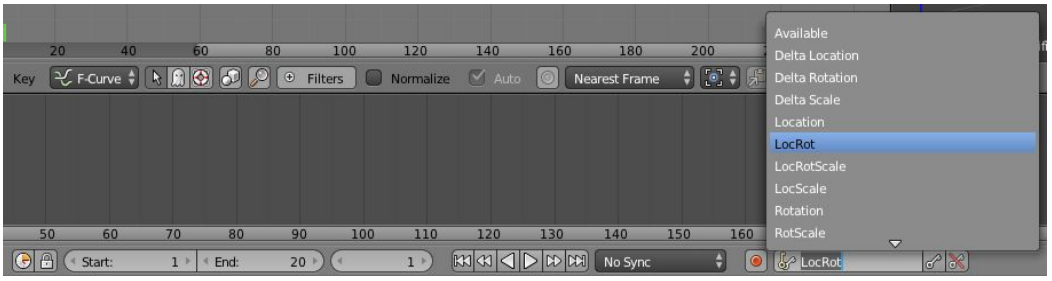
b. Add a new action by clicking on the button labeled *New* and rename the name *Action* to *Wave*.

c. Click on the *F* on the right side of the action's name. *F* forces that the action is saved within the blend file.



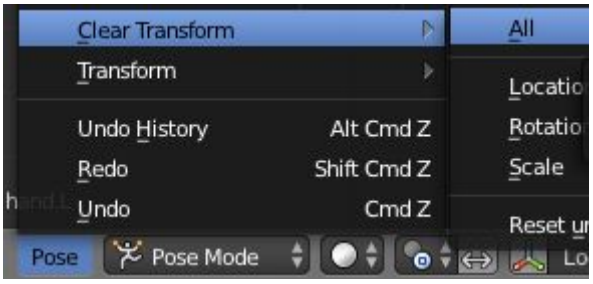
d. On the bottom, *Timeline* bar find the *Active Keying Set* field and set it to *LocRot*. The field has an icon of two keys in it. It guarantees that we only modify the location and rotation of the bones.

e. Also on the bottom *Timeline* bar edit the start and end frames for the animation set them to 1 and 20 respectively.



9. **Add a clean T-Pose to frame 0**: In *Pose Mode* we add a clean pose with no animation to frame 0.

a. Move the timeline to frame 0. Select all bones with [A] and choose:



b. With the cursor over the *3D viewport* or the *Dope Sheet* press [I] to add a keyframe for all bones at frame 0.

10. **Animate in *Pose Mode***: We can finally create our first animation by rotating the arm and hand bones for a waving animation.

a. Set the green line in the Dope Sheet to frame 1. Move [G] and rotated [R] the hand into a beginning waving pose.

b. Select all bones with [A] and add a keyframe with [I]. A keyframe will be added for all selected bones and you should see it in the action editor.

c. Set the green line in the Dope Sheet to frame 20 and the same keyframe again [I].

d. Move the green line to frame 10.

e. Move the hand to the second wave position and add a keyframe [I].

f. Press *Play* in the *Timeline* bar to run the animation. The animation will interpolate now between the 3 keyframes. You will see that a natural looking wave animation involves many bones of the body, not just the bones of one arm.

Rigify to Unity script source is this tutorial video:
https://www.youtube.com/watch?v=lAwK19Ijq50

Checkpoint 2: Rigged & Waving Sintel

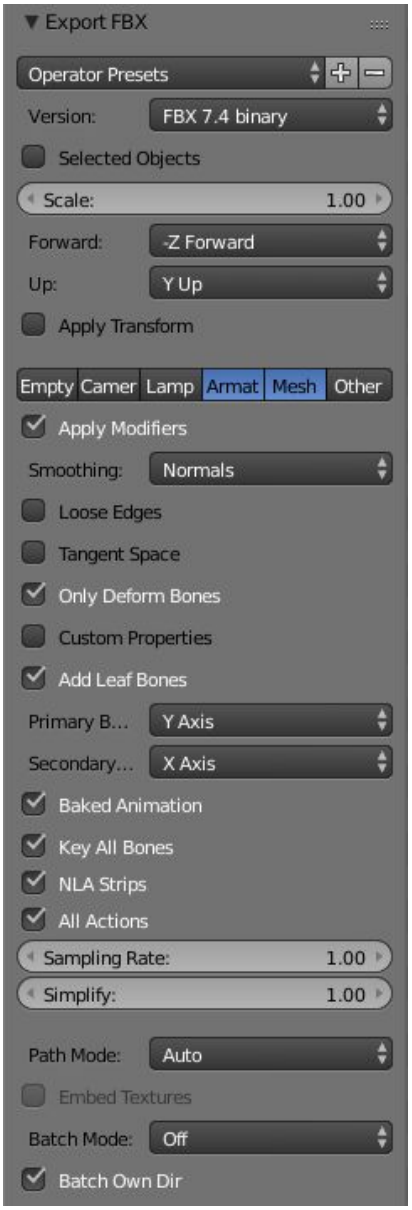
2.3 Unity Character Import via FBX Export & Import

2.3.1 Blender Export settings

There are multiple ways of bringing our Blender creations into Unity. We could either import in Unity a .blend file directly without a special Blender export. But Unity will do behind an FBX-file conversion first on which we don't have any influence. If you want to configure the FBX-export settings you must explicitly export for the FBX format. The advantage of a direct blend-file import is you can continue editing in Blender while a blend-file is already within the assets folder. Unity will automatically import the changes.

2.3.1.1 Blender Export as FBX

1. Go to *File* > *Export* and choose .fbx
2. In the bottom left you will find a list of export options with the header *Export FBX* use the same settings you see in the screenshot below:



2.3.2 Import Blend file or FBX with handmade Rig in Unity

1. **Create Unity Project & Import Character**:
   a. Start Unity and create a new project named *2.3 Unity Character Import*.
   b. Save the scene with *File* > *Save Scene* as *Scene1*.
   c. Copy the FBX file from the Blender export into the Asset folder of the Unity project. Copy also the texture file that was used in Blender.
2. **Set Character Properties**: With the selected character in the *Project* window you can inspect its properties in the *Inspector* windows. It has 3 tabs:
   a. ***Model***: General Settings, leave them as they are

b. ***Rig***: Set the *Animation Type* to *Humanoid* and press *Configure*, if everything is green the humanoid rig in T-pose was correctly recognized. Press *Done* to return to the *Rig* screen. A checkmark left of the *Configure* button show successful rig interpretation.

c. ***Animations***:

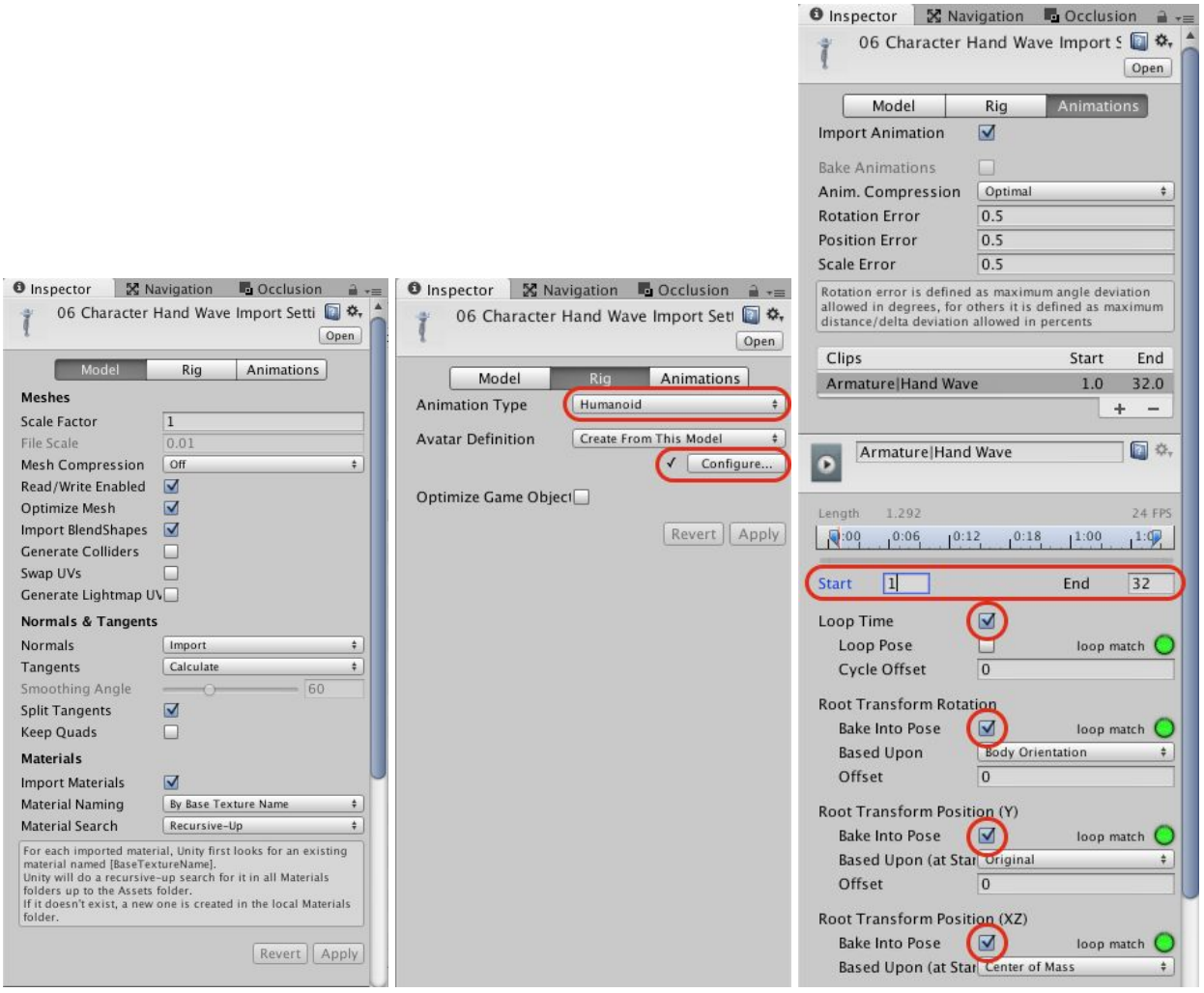
   i. ***Animation Compression***: Leave this parameter on *Optimal*. If an animation contains many keyframes the amount of data can be large. In the worst case, every frame can be a keyframe. Unit tries to compress this information by reducing keyframes that are insignificant for the animation.

   ii. Set the ***Start*** frame to 1 and the ***End*** frame to 32. In the Clips table above you could define additional animations between different start and end frames.

   iii. Check the ***Loop Time*** option: If checked the animation will loop endlessly.

   iv. Check the ***Loop Pose*** option: If checked Unity tries to fix differing transforms of the start and end frame. The green lamps show that the start and end frame match.

   v. The ***Root Transform*** is a projection on the Y plane of the body transform and is computed at runtime. At every frame, a change in the root transform is computed. This change in transform is then applied to the game object to make it move. The settings - *Root Transform Rotation*, *Root Transform Position (Y)* and *Root Transform Position (XZ)* - let you control the root transform projection from the body transform.
   Check the ***Bake into Pose*** options if you don't want to influence the root motion by that part of the body transform. See the Unity docs for more details.



2.3.3 Importing Animations as separate FBX files

The animations provided by us for this course come in separate FBX files. Those FBX files also contain a skeleton each, which is referenced by the animation. One of the strengths of Mecanim is to simply use humanoid animations on any humanoid character. The thing to remember when using separate files for model and animation is to set the *Animation Type* in the import settings to *Humanoid* for both, otherwise, the animations won't work correctly.

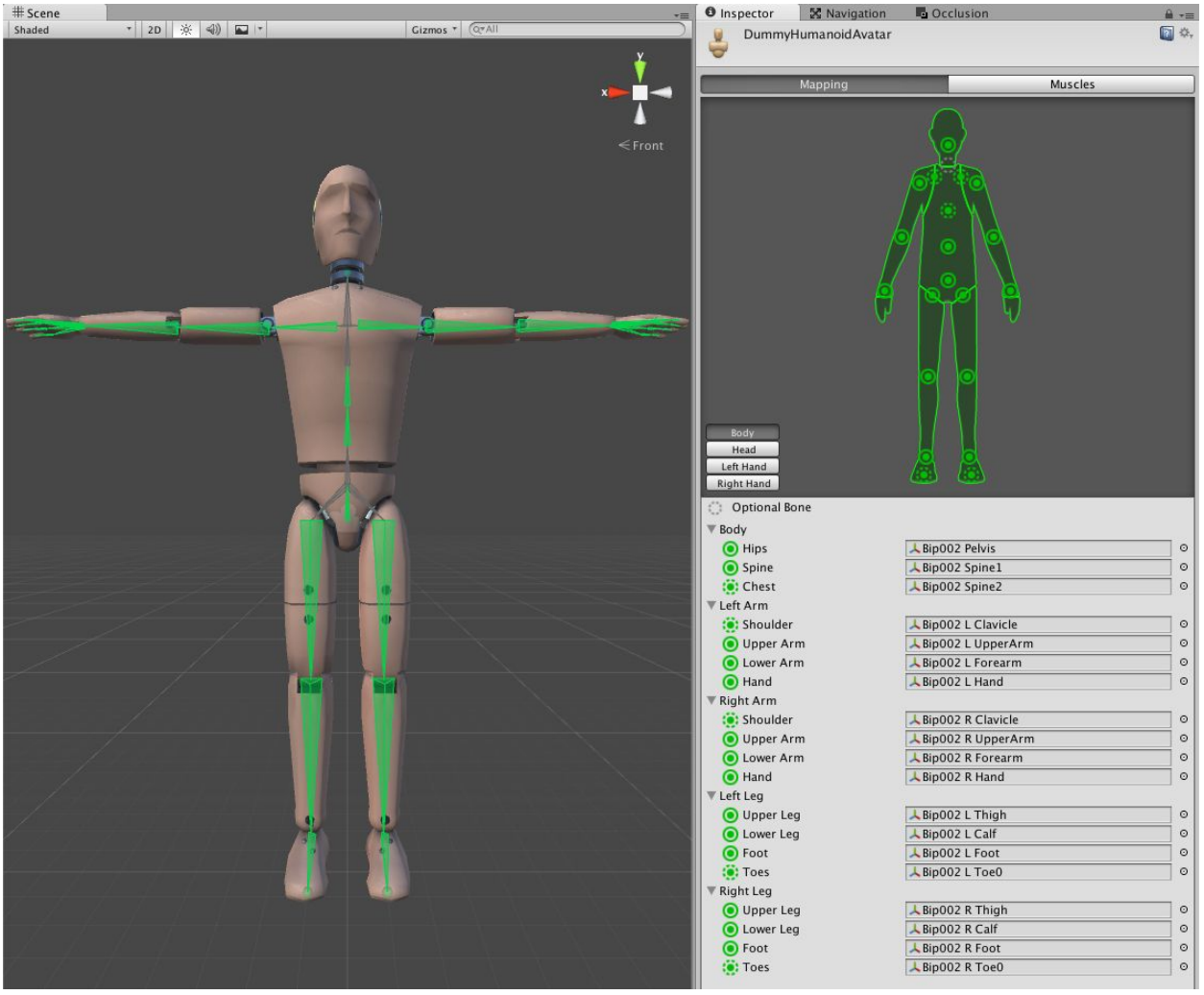
2.3.4 Unity's Humanoid Rig and the Muscle Space

2.3.4.1 Unity's Humanoid Rig

In the mapping screen of the *humanoid* configuration, you see that Unity tries to maps the different bones of the imported rig to the Unity humanoid rig with specific bones. It can be that the imported rig has in certain areas more or fewer bones. Unity's humanoid rig has the following specific definitions:

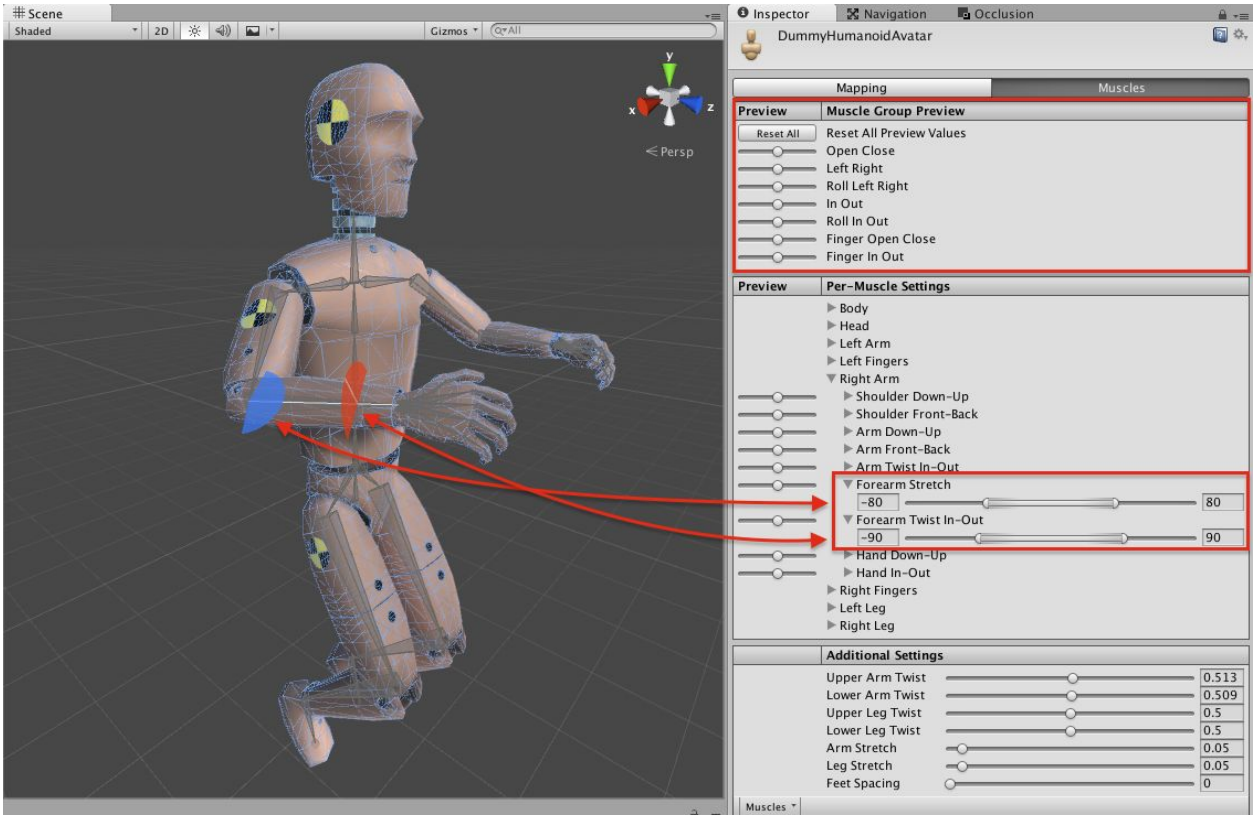
- 2 spine bones
- 1 neck bone
- only rotations
- no twist bones with different rotations at the ends of the bone
- optional bones: *chest*, *shoulders*, *toes, eyes, jaw*

If Unity cannot map the imported rig to its rig it will indicate the missing bones in red color.

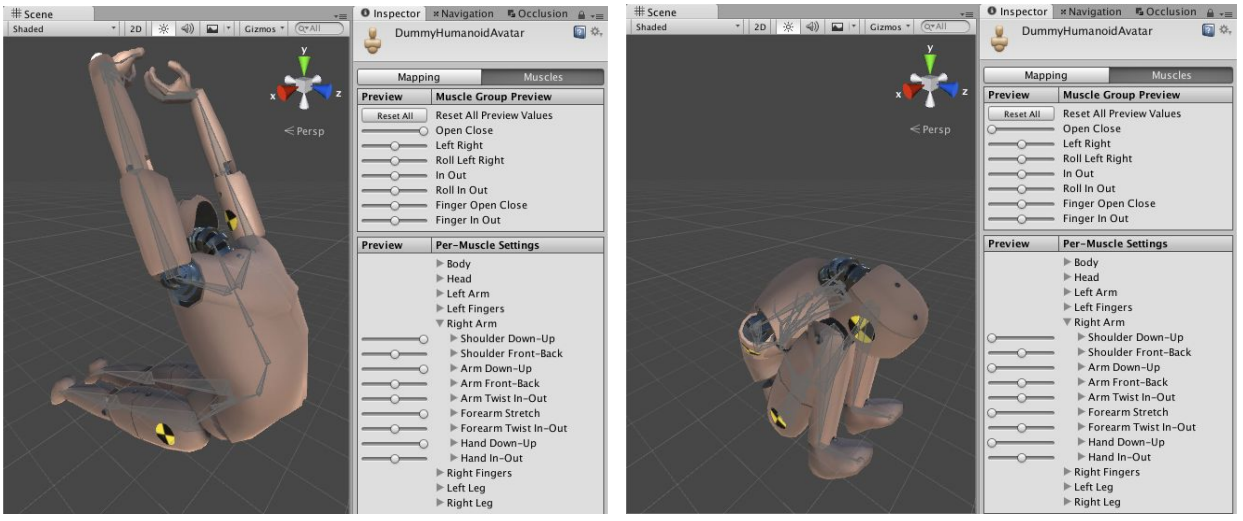


2.3.4.2 Unity's Muscle Space

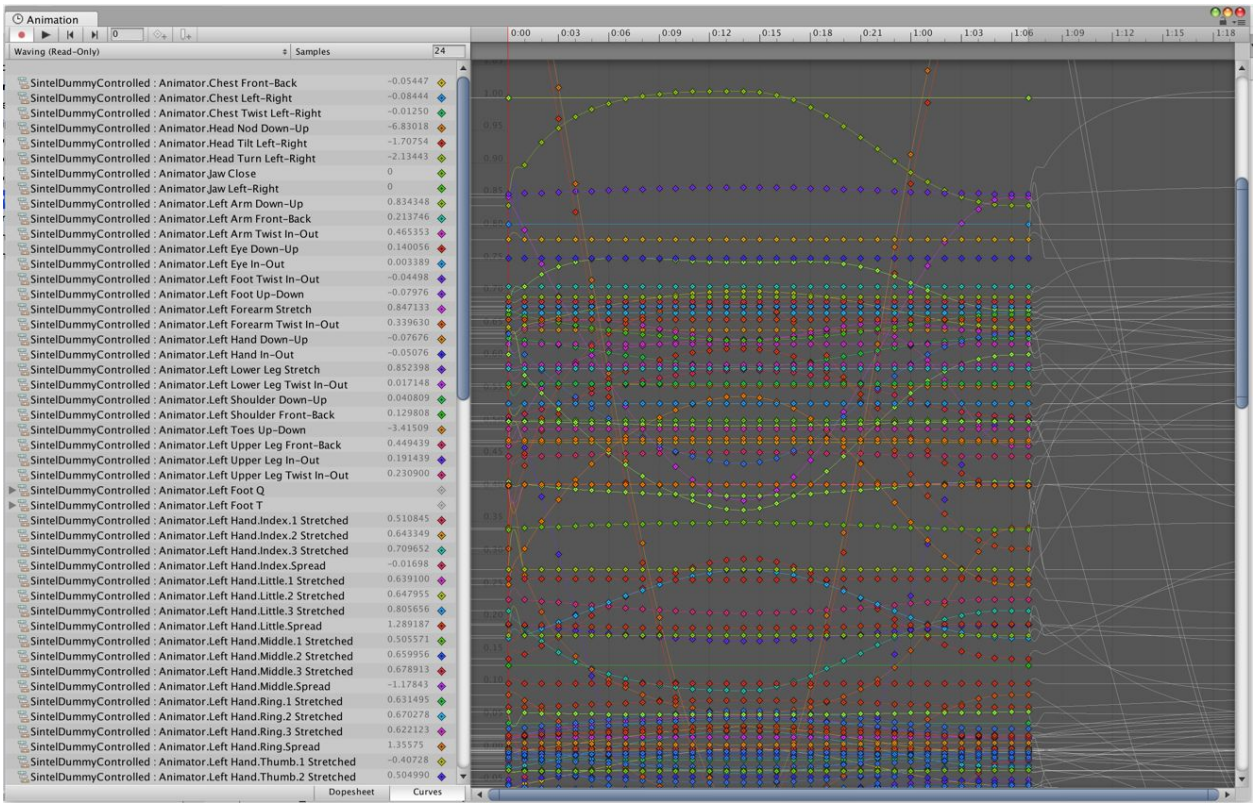
Unity creates after the bone mapping it's so-called *Muscle Space* where it maps the bones to 47 *Muscles*. For each muscle, it defines possible rotations around each axis. As in the following screenshot depicted is e.g. the *Forearm Stretch* restricted to a range of -80° to +80° on the y-axis (blue angle). This allows the restriction of joint rotation to the real limitations. The red angle shows the allowed *Forearm Twist In-Out* (x-axis). All the allowed ranges are then normalized to the range -1 to 1.



Unity defines furthermore *Muscle Groups* that define full body motions with groups of bones. With a slider, you then can see what is possible with the normalized range of the included bones.



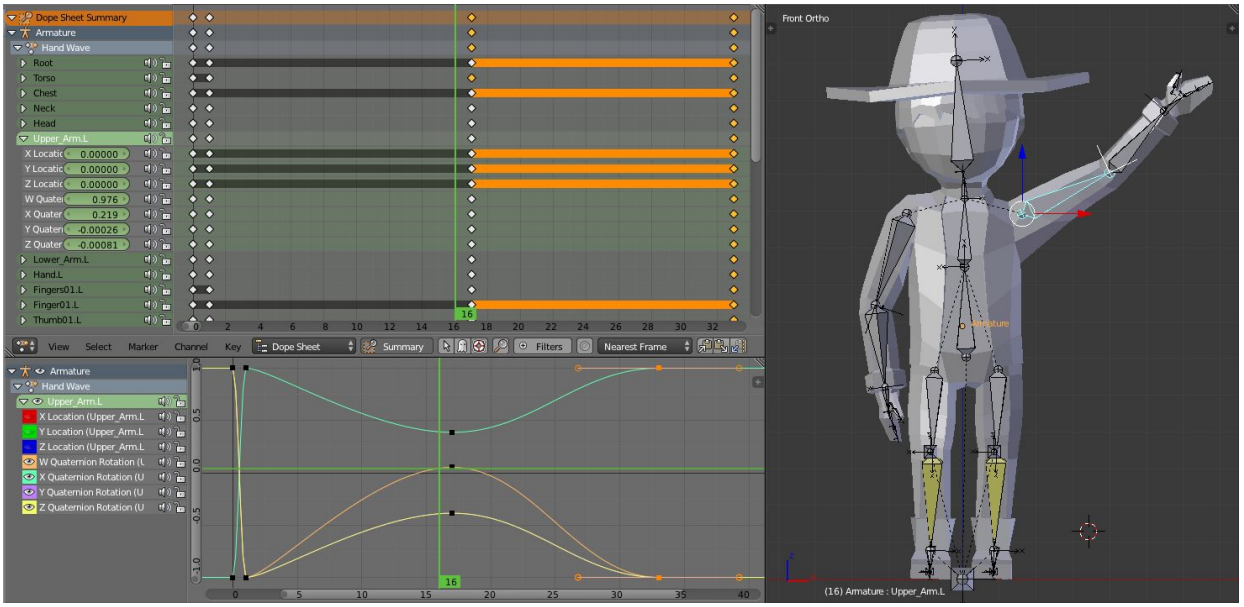
Unity will now map all animations to this muscle space. You can see that in the *Animation* window of the selected Sintel character. If you select the *Waving* animation you can inspect all interpolation curves of all muscles.



[Robertl writes in the Unity blog](#) with some of the rare insights on Mecanim the following:

One beautiful thing about Muscle Space is that it is completely abstracted from its original or any skeleton rig. It can be directly applied to any Humanoid Rig and it always creates a believable pose. Another beautiful thing is how well Muscle Space interpolates. Compare to standard skeleton pose, Muscle Space will always interpolate naturally between animation keyframes, during state machine transition or when mixed in a blend tree.

If we compare this to what we have made in *Blender* for the *Waving* animation, we see that *Blender* works internally with Quaternions wheres as Unit's muscles correspond to the Euler angles of the joints. We also see that the bones axis are not the same anymore as we've defined them in *Blender*.



2.4 Introduction to Unity Mecanim

State Machine: Mecanim is Unity's underlying animation state machine. It can be used to switch animations or blend between them in various different ways. Mecanim also provides inverse kinematics for biped characters.

Animate any public value: Mecanim is not only for animating humanoid biped characters, but it can also be used for animating UI Elements, opening doors, changing lights.
Any publicly exposed value can be animated with Mecanim.

There are many video tutorials on Mecanim. We can recommend the following:

Unity: Mecanim Animation Tutorial

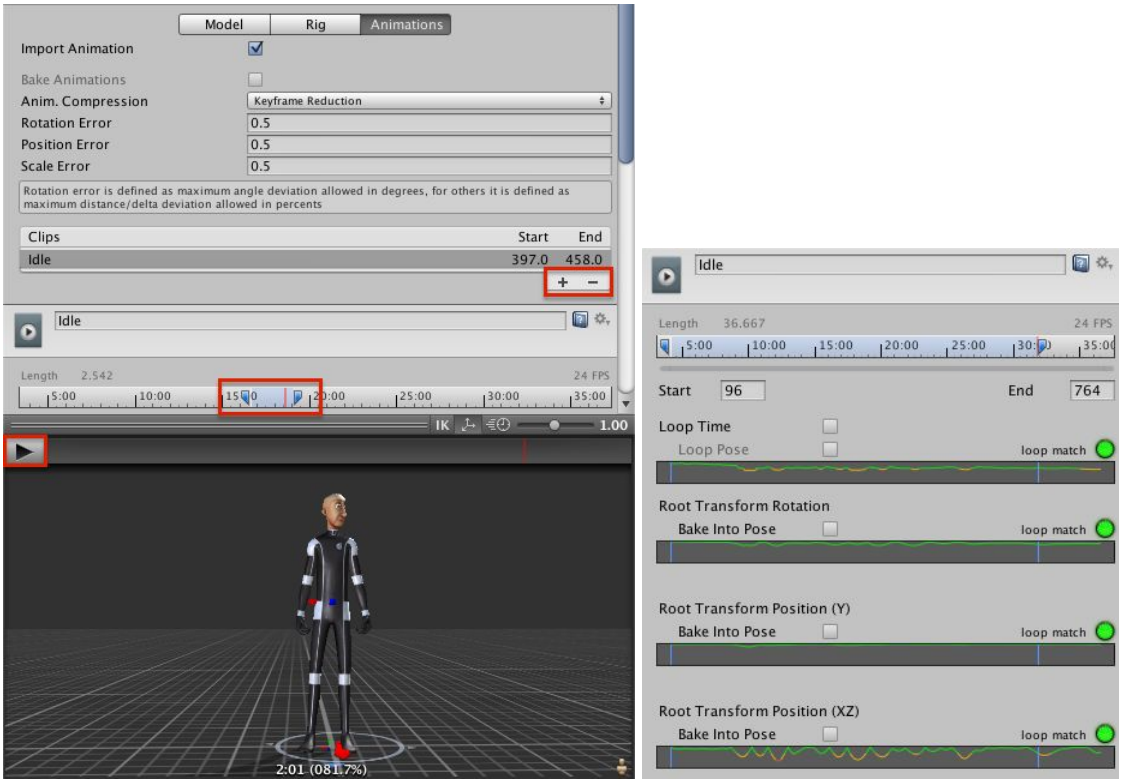
Mixamo: Unity Animation Overview

Mixamo: Advanced Mecanim Animation

2.4.1 Apply an Animation with an Animator Controller

As an introduction to mecanim, we will create a simple Animator Controller that utilizes two animation clips. Add the idle and walking animations into your project.

1. **Copy the last Sintel project** and name it *2.4 Mecanim Tutorial*.
2. **Copy Animations and Character**: Copy the following folders from the DropBox into the project's *Assets* folder:
   a. *02 Rigged Skeleton Animation/2.4 Mecanim Tutorial/Animations*
   b. *02 Rigged Skeleton Animation/2.4 Mecanim Tutorial/Characters*
3. **Place Character in the Scene**:
   a. Check the import settings for the rig of the *Assets/Characters/FinalIK-Dummy/DummyHumanoid* character. Make sure all bones were recognized.
   b. Drag the *DummyHumanoid* character into the scene.
   c. Set its position to [0,0,0] and move the camera in front of it.
   d. Save the scene.
4. **Adjust the Animation**: In the *Animation* tab of the import settings, you can preview the animation with a generic avatar. You can see that the imported animation shows only a small portion of the entire motion-captured animation.
   a. Select the *HumanoidIdle* animation in the *Assets/Animations* folder.
   b. Choose the start frame 120 and an end frame beyond 700.
   c. Check that all loop math lamps are **green**. A green lamp indicates a good match from end to start.
   d. You can enlarge the animation preview window in a separate window by right-clicking title bar of the preview subwindow.

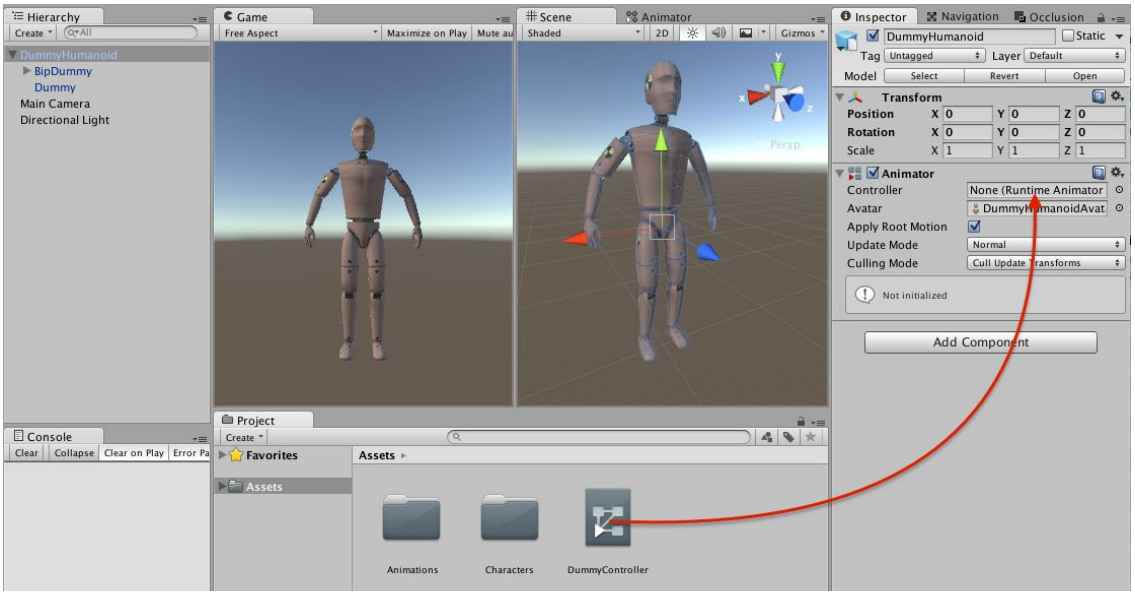


If the animation doesn't work, check again that have set the *Animation Type* for the Rig to *Humanoid*.

5. **Add an Animator Controller**:

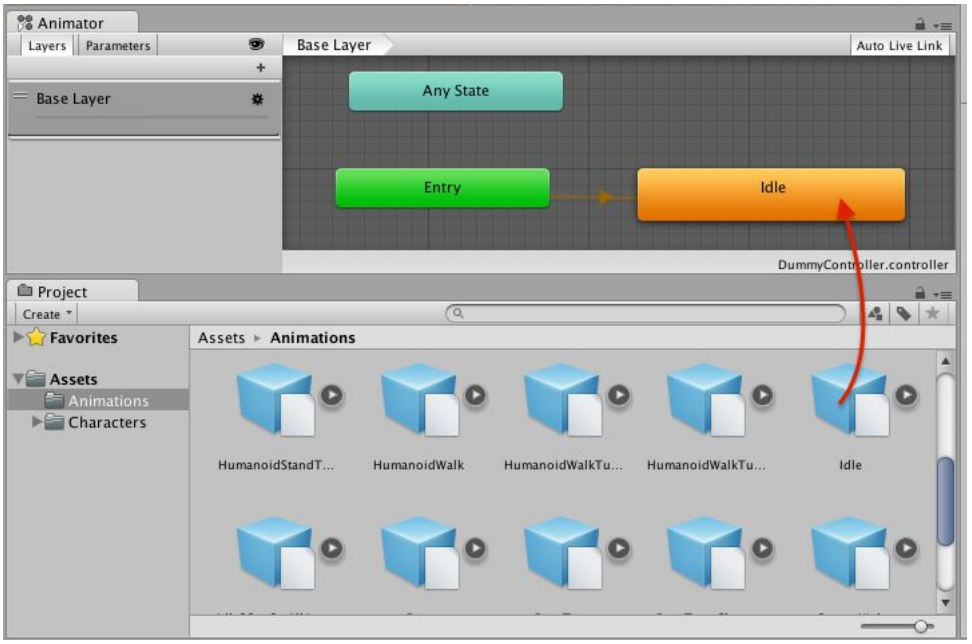
   a. Create a folder named *Animators*. In there create an animator controller with *right-click > Create > Animator Controller* and name it *DummyController*.

   b. In the *Hierarchy* window select the character and drag the *DummyController* onto the *Controller* field in the *Inspector*.



6. **Add the Animation to Animator**: Double-click the *DummyController* to open the *Animator* window and drag the *HumanoidIdle* animation from the *Animations* folder onto

the *Animator* window.



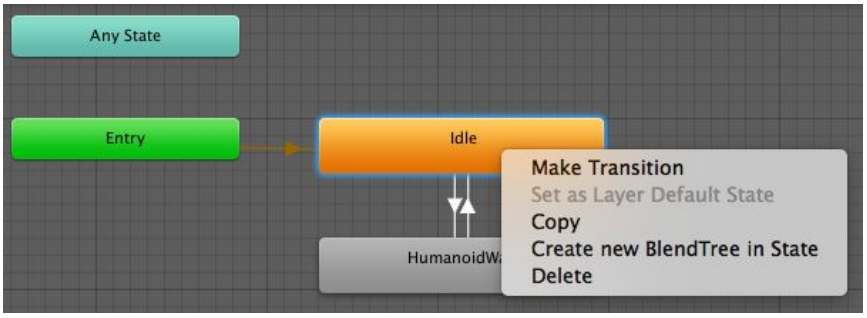
7. **Play**: Press *Play* to see the *HumanoidIdle* animation in action.

2.4.2 Add Transition in between Idle and Walk Animation

Now lets to our first state change in the Animator window.

1. **Add Walk Animation**: Drag the *HumanoidWalk* animation into the Animator window. The new state will have a gray color. The orange color is for the initial default state.

2. **Make Transitions**: Create a transition arrow between the *HumanoidWalk* and the *HumanoidIdle* animation by right-clicking and select *Make Transition* and drag. Create a second transition arrow in the opposite direction:



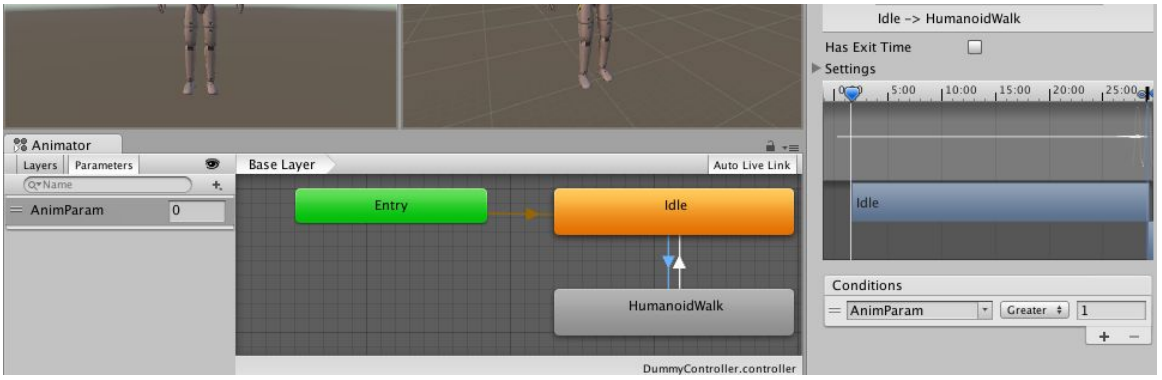
3. **Add Animation Parameter and Transition Conditions**:

   a. Click on the *Parameters* Tab in the *Animator* window.

   b. Add an Int parameter with the +-button and name it *AnimParam*.

   c. Select the transition from *HumanoidIdle* to *HumanoidWalk*.

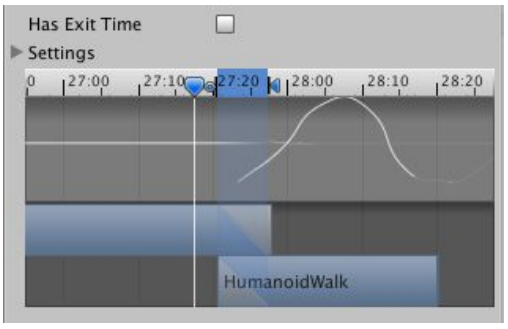
d. Add a condition with the +-button and set *AnimParam equals 1*



e. Select the transition from *HumanoidWalk* to *HumanoidIdle*.

f. Add a condition with the +-button and set *AnimParam equals 0*

4. **Fine tune a transition**: You can precisely configure a transition in the Inspector.
- The overlap region of the two involved animations can be zoomed with the mouse wheel.
- You can adjust the amount of overlap by dragging the lower animation bar.
- You can set the start and the end frame of the transition.



5. **Add a Player C#- Script**: The setting of the AnimParam condition from 0 to 1 has to be done in a script:

a. In the Project window add a new C# script and name it *PlayerBehaviour*.

b. Write the following code:

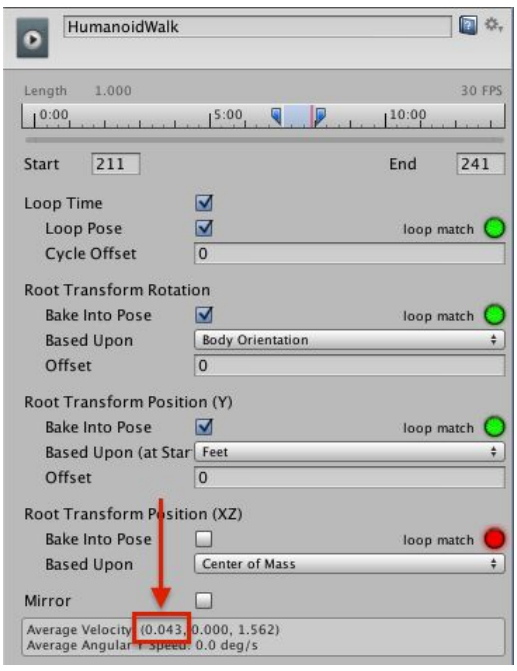
```
using UnityEngine;
using System.Collections;

public class PlayerBehaviour : MonoBehaviour {

    private Animator _animator;

    void Start () {
        _animator = GetComponent<Animator>();
    }

    void Update () {
        if (Input.GetKey("up"))
            _animator.SetInteger("AnimParam",1);
        else _animator.SetInteger("AnimParam",0);
    }
}
```

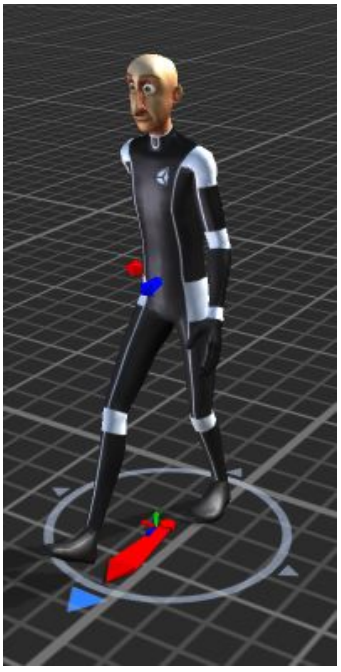
c. Drag the script onto the *DummyHumanoid* game object.

6. **Play**: Press play and see how the dummy starts walking when you press the up button. If the character doesn't start immediately with walking, you may have set the option *Has Exit Time*. Uncheck this option.

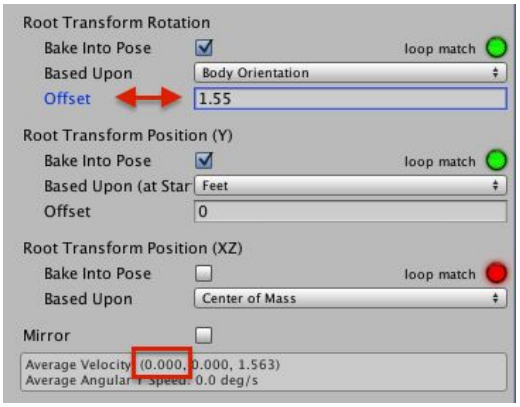
Checkpoint 3: Idle-Walking Controller

7. **Correct Walk Direction**: You may notice that your character is not walking perfectly straight. You can see that best in the animation preview in the *Inspector* window. If you start the animation the character starts on the line and after a while he is not anymore on the line.
You can also see this in the x- and y-values of the *Average Velocity* vector:





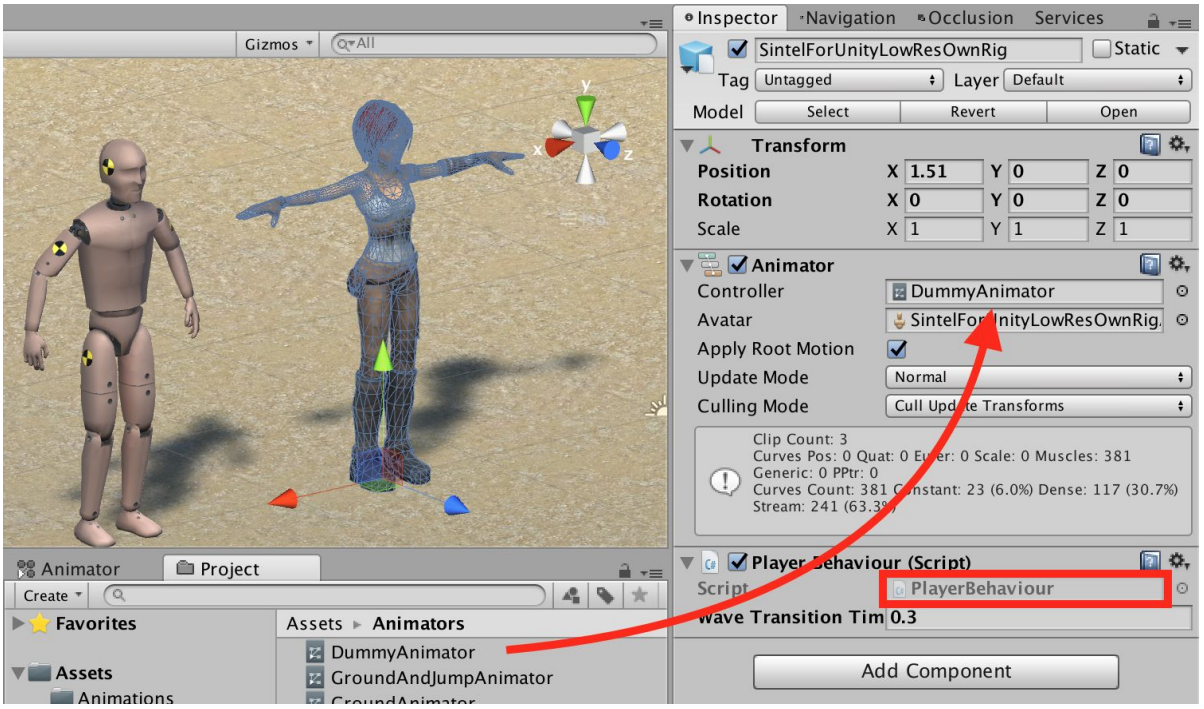
You can correct this by adding an offset to the root transform rotation. Do that by sliding between the *Offset* label and the offset value box until you have 0.000 in the x-value of the *Average Velocity* vector:



2.4.3 Apply the Animator Controller to another Character

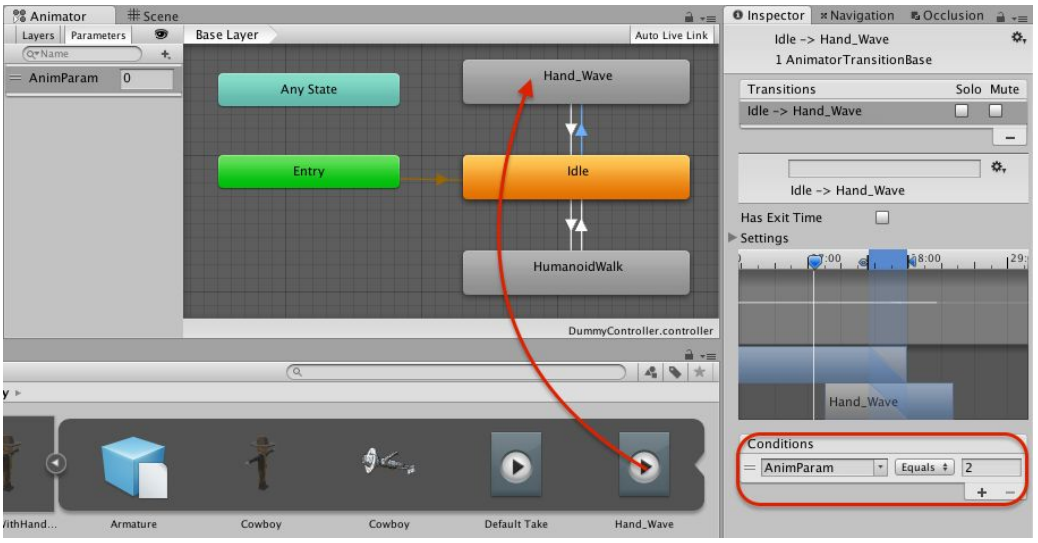
To see the power of Mecanim we want now to apply our animator controller to the Sintel character that has a different rig and was modeled in a different tool than the dummy character.

1. **Place the Sintel Character**:
   a. Drag your *Sintel* character into to scene view and rename it *SintelDummyControlled*.
   b. Place it at [0,0,0] and move it beside the Dummy character.
2. **Add Controller & Script**: Drag the *DummyController* and the *PlayerBehaviour* script onto the Sintel game object.



3. **Let the camera move with a character**: For not walking out of the cameras view, simply add the camera to one of the characters.
4. **Play!** The Sintel character should play now the same animation.
5. **Add Wave Animation and Transition Conditions**:
   a. Add your wave animation from within the Sintel character to the *Animator* window.
   b. Add two transitions from and to the *Idle* animation.
   c. Add the condition *AnimParam equals 0* for stopping the hand wave.

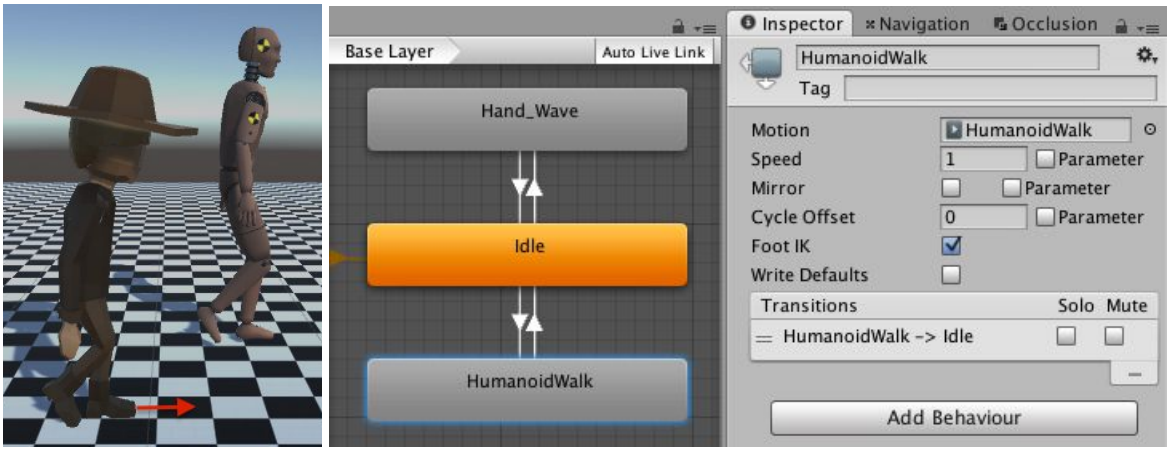
d. Add the condition *AnimParam equals 2* for starting the hand wave.



6. Add the additional state change in the PlayerBehaviour script:

```
// Update is called once per frame

void Update () {
    if (Input.GetKey("up"))
        _animator.SetInteger("AnimParam",1);
    else if (Input.GetKey("space"))
        _animator.SetInteger("AnimParam",2);
    else
        _animator.SetInteger("AnimParam",0);
}
```

7. **Avoid Foot Slipping for smaller Characters**: If set a chessboard pattern as a ground plane you will notice that the smaller characters feet slip slightly on the ground. This is due to the different sizes and leg ratios of the different bodies.
You can avoid that by checking the *Foot IK* option in the *HumanoidWalk* state properties in the *Inspector* window:



8. **Play!**: Press play & see how the characters move absolutely in sync. even though that:
- the rigs of the characters are different
- the mesh and the vertex weights are different
- the animations were designed in different tools on different rigs.

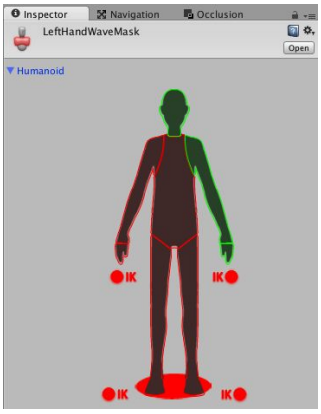
2.4.4 Combine Multiple Animations with Layers & Avatar Masks

Now we can let the character idle, wave or walk. But what if we want to wave the hand while walking? We can achieve that by using another layer in the *Animator* window and by using an *AvatarMask* to restrict the waving animation to the left arm and hand.

1. **Create an Avatar Mask**:

   a. In the *Animators* folder right-click and choose *Create* > *AvatarMask* and name it *LeftHandWaveMask*.

   b. Select it and open the *Humanoid* dropdown in the *Inspector* window. Disable all body parts but the head and the left arm and hand. (Left in terms of the characters view.)



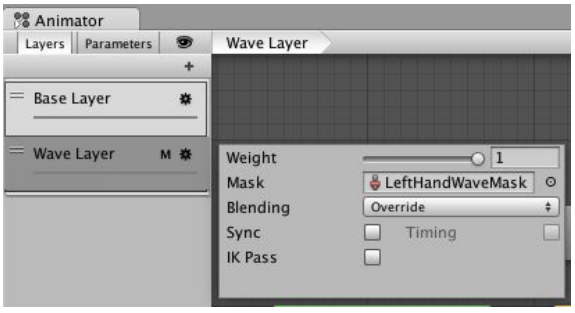
2. **Create a new Layer**: Select the *Layer* tab in the *Animator* window and create a new layer by clicking the +-sign. Name it *Wave Layer*.

   a. Add the *Hand_Wave* as the only state on this layer.

   b. Click on the settings wheel of the new *Wave Layer* and:

      i. Set the *LeftHandWaveMask* as the mask

      ii. Set *Override* for Blending



3. **Correct the Base Layer**: We don't need the *Hand_Wave* animation anymore in the Base Layer. Remove it by *right-click* > *Delete*.

4. **Adapt the *PlayerBehaviour* Script**: Now our character can wave and walk, so we need to separate if statements:

```
void Update()
{
    if (Input.GetKey("up"))
        _animator.SetInteger("AnimParam", 1);
    else _animator.SetInteger("AnimParam", 0);
```

```
        if (Input.GetKey("space"))
             _animator.SetLayerWeight(1, 1.0f);
        else _animator.SetLayerWeight(1, 0.0f);
    }
```

5. **Smoothing out the interpolation:** We are now able to walk and simultaneously wave our hand by holding down space. But then pressing and releasing space the wave arm jumps instantly from one position to the next. To solve this we can manually interpolate the layer weight in our script:

```
public class PlayerBehaviour : MonoBehaviour
{
    public float        transitionTime = 0.3f;
    private Animator    _animator;
    private float       _waveWeight = 0.0f;
    private float       _deltaT;

    void Start()
    {   _animator = GetComponent<Animator>();
        _deltaT = 1.0f / transitionTime;
    }

    void Update()
    {
        if (Input.GetKey("up"))
             _animator.SetInteger("AnimParam", 1);
        else _animator.SetInteger("AnimParam", 0);

        if (Input.GetKey("space"))
        {   if (_waveWeight < 1.0f)
                _waveWeight = Mathf.Clamp(_waveWeight +
                                          _deltaT * Time.deltaTime,
                                          0.0f, 1.0f);
            _animator.SetLayerWeight(1, _waveWeight);
        } else
        {   if (_waveWeight > 0.0f)
                _waveWeight = Mathf.Clamp(_waveWeight -
                                          _deltaT * Time.deltaTime,
                                          0.0f, 1.0f);
            _animator.SetLayerWeight(1, _waveWeight);
        }
    }
}
```

Note: *We are using a simple linear interpolation in the script shown above. Using linear interpolations for animation transitions often leads to unsatisfying results that look unnatural because of a lack of continuity. Using animation curves for these kinds of interpolations will result in much more pleasing animations.*

6. **Play!** The characters should lift now their hands more slowly.

2.4.5 Walking and Run Animator with Blend Trees

We will extend now the animator controller with so-called *blend trees*. They allow a more complex blending of different states.

Blending one integer dimension: So far we blended one integer parameter and the in-between transition was handled by the blend in the timeline overlap. We could also say that we blended with the *AnimParam* and the up/down-keys one dimension.

Blending two float dimensions: Now we want to combine the up/down- with the left/right keys to smoothly blend the forward with the left or right turn animation.
In addition, we want to differentiate the dimensions with float parameters to allow in-between states. This, of course, can only be used on input devices that allow in between states such as gamepads. On keyboards the up/down- and left/right-keys only allow the values -1, 0 and 1.

1. **Drag another Sintel Character into the Scene** beside the others and rename it *SintelBlendTreeControlled*.

2. **Create a new animator controller** named *WalkAndRunAnimator* and assign it to your character.

   a. Add two float parameters called *Forward* and *Turn*.

   b. Right-click on the work area and choose *Create State > From New Blend Tree*.
      - Name it *Walk and Run* in the Inspector.
      - Set *Foot IK* to true

3. **Configure the blend tree**: Double-click on the new state to open it.

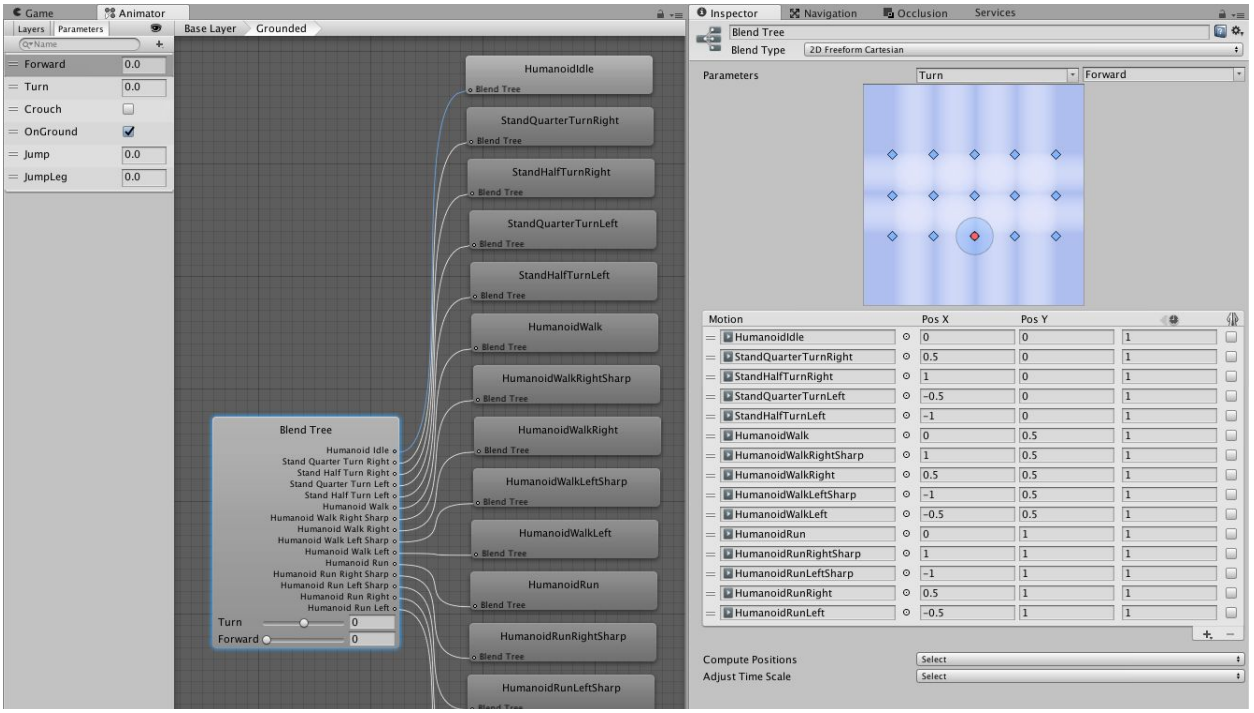
   a. Change the blend tree's blend type to *2D Freeform Cartesian*

   b. Choose for the first parameter the *Turn* variable.

   c. Add 15 motion fields to the blend tree like in the image below with the +-sign.
      We will add
      - 5 animations for standing with a y-value (speed) of 0.0
      - 5 animations for walking with a y-value (speed) of 0.5
      - 5 animations for running with a y-value (speed) of 1.0

      Drag the listed animations & write the according to parameter values as in the next image.

   d. You can move around the red point in the blueish blend plane with or without playing the animation.
      - Moving horizontally in the **bottom** row interpolates all standing animations.
      - Moving horizontally in the **middle** row interpolates all walking animations.
      - Moving horizontally in the **top** row interpolates all running animations.
      - Moving vertically interpolates the animations between standing, walking & running.



4. **Create a new control script**:
   Create a new C# script *PlayerBlendedBehaviour* that controls the forward and turn parameter based on user key presses. It is advised to use smoothed values for the turn and forward so that the animations continuous and not janky looking.

```
public class PlayerBlendedBehaviour : MonoBehaviour
{   private Animator _animator;

    // Use this for initialization
    void Start () {
        _animator = GetComponent<Animator>();
    }

    // Update is called once per frame
    void Update () {
        float walkOrRun = Input.GetKey(KeyCode.LeftShift) ? 0.5f : 1.0f;
        _animator.SetFloat("Forward", Input.GetAxis("Vertical") * walkOrRun);
        _animator.SetFloat("Turn",    Input.GetAxis("Horizontal"));
    }
}
```

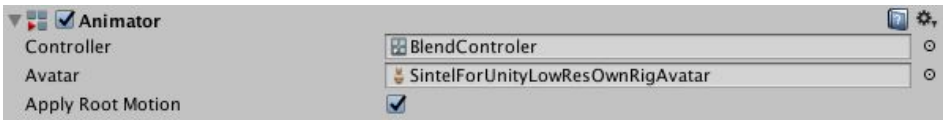
5. **Add a better camera script**: You can add the smooth follow camera script from the last semester or add the *MultipurposeCameraRig* from the Standard Assets/Cameras/Prefabs.
   - Remove the Camera game object.
   - Set the character as the target of the new camera.

6. **Apply the blend controller and the script**: Drag & drop the *BlendController* and the *PlayerBlendedBehaviour* script to the *SintelBlendTreeControlled*.

Checkpoint 4: Blend Tree

2.5 Setting up a Character Controller

Now that we know how to use Animator Controllers it is time to bring everything together by creating a controllable character. At the moment our *WalkAndRunAnimator* moves the character because we have set to apply the root motion of the animation.



This means that character is moved by the motion within the animation. This works because we did not bake the *Root Transform Position (xy)* in the animations settings.

Our character can therefore not fall to the ground nor can it react to ramps or obstacles. It has no rigidbody component attached and is not attracted by the earth's gravity.

You can add a cube, scale and rotate it to a ramp and place in front of the character. The character will just walk through the ramp:

There are in general two possibilities to implement a character controller:

- **With the *Unity Character Controller* component**: This is for third-person or first-person player control that does not make use of rigidbody physics. If not programmed especially all motion changes occur immediately and therefore not realistic.
- **With a rigidbody and a capsule collider**: Such type of controllers can be told to move in some direction from a script. The controller script will then carry out the movement but are constrained by collisions. It will slide along walls, walk upstairs and walk on slopes.

You can find a more detailed discussion of the differences in this video by Master Indie.

2.5.1 Character Controller with Physics

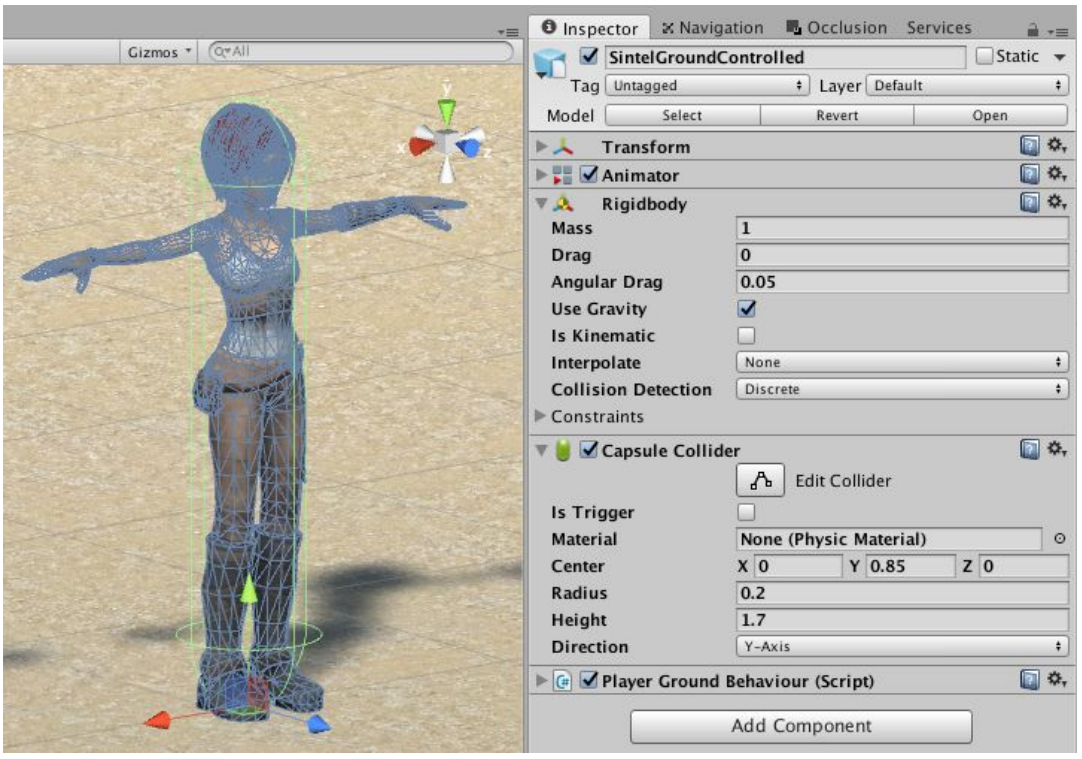
A character controller with physics has clearly more advantages and possibilities than the *Unity Character Controller*. We will develop such a controller in two steps.

In the first step, we create a controller with our Walk and Run Animator that can walk up ramps and that falls back on the ground again.

In the second step, we extend this controller so that it can jump up and down.

2.5.1.1 Controller with Walk and Run Motion

1. **Copy the previous project** and rename it to *2.5 Character Controller*.
2. **Drag another Sintel Character into the Scene** beside the others and rename it *SintelWalkAndRunControlled*.
3. **Assign the previous animator controller** named *WalkAndRunAnimator*.
4. **Add a Rigidbody and a Capsule Collider component** and adapt colliders *Center*, *Radius* and *Height* as follows:



5. **Create and assign a new Script and name it *PlayerWalkAndRunBehaviour***:
   The big difference is that we now have to move the character by setting the rigidbody's velocity. We do that by overriding the method ***OnAnimatorMove***. This will automatically disable the *Apply Root Motion* option and marks it as *Handled by Script*. The script has the following structure:

```
public class PlayerWalkAndRunBehaviour: MonoBehaviour
{
    public float        turnSpeed = 360;
    public float        forwardSpeed = 1.0f;

    private Rigidbody   _rigidbody;     // Reference to the characters rigidbody
    private Animator    _animator;      // Reference to the characters animator
    private Transform   _cam;           // Reference to the main camera of scene
    private Vector3     _moveDir;       // Move direction derived from inputs
    private float       _turnAmount;    // Amount of forward movement
    private float       _forwardAmount; // Amount of turn movement

    private void Start() {...}
    private void FixedUpdate() {...}
    private void Move(Vector3 moveDir) {...}
```

```
    public void OnAnimatorMove() {...}
}
```

a. In the **Start** method we get some references to components:

```
private void Start()
{   _animator = GetComponent<Animator>();
    _rigidbody = GetComponent<Rigidbody>();

    // Avoid rotation on rigidbody. The capsule will be always upright.
    _rigidbody.constraints = RigidbodyConstraints.FreezeRotationX |
                             RigidbodyConstraints.FreezeRotationY |
                             RigidbodyConstraints.FreezeRotationZ;

    // We will use the camera's orientation to determine the walk direction
    // The main camera object must be tagged with the tag MainCamera
    if (Camera.main != null)
        _cam = Camera.main.transform;
    else
        Debug.LogError("Warning: no main camera found.", gameObject);
}
```

b. We do all updates from within the **FixedUpdate** method to be in sync with the physics engine.

```
private void FixedUpdate()
{
    // Read inputs
    float inputH = Input.GetAxis("Horizontal");
    float inputV = Input.GetAxis("Vertical");

    // Calculate move direction from the camera orientation
    _moveDir = inputV*_cam.forward*forwardSpeed + inputH*_cam.right;

    // We scale down the move vector for walking
    if (Input.GetKey(KeyCode.LeftShift)) _moveDir *= 0.5f;

    Move(_moveDir);
}
```

c. In the **Move** method we calculate the amount of forward and turn motion, so that we can pass them at the end to the animator. We also apply the turn amount as a rotation to the game object.

```
private void Move(Vector3 moveDir)
{
    // Convert the move vector from world to local coords.
    moveDir = transform.InverseTransformDirection(moveDir);

    // Take the angle between the z-axis and the x-axis as turn amount
    _turnAmount = Mathf.Atan2(moveDir.x, moveDir.z);

    // Take the z-axis as forward amount
    _forwardAmount = moveDir.z;

    // Apply rotation because the rigidbody rotation are freezed
    // The position will be set by physics later in the OnAnimatorMove method
    transform.Rotate(0, _turnAmount * turnSpeed * Time.deltaTime, 0);
```

```
        // Update the animator parameters
        _animator.SetFloat("Forward", _forwardAmount, 0.1f, Time.deltaTime);
        _animator.SetFloat("Turn", _turnAmount, 0.1f, Time.deltaTime);
    }
```

d. In the **OnAnimatorMove** callback method we reverse calculate a velocity from the animators delta position divided by the delta time and apply it to the rigidbody:

```
public void OnAnimatorMove()
{
    if (Time.deltaTime > 0)
    {
        // On ground we calculate the speed from the root motion
        Vector3 v = _animator.deltaPosition / Time.deltaTime;

        // We preserve the existing vertical part of the current velocity
        v.y = _rigidbody.velocity.y;

        _rigidbody.velocity = v;
    }
}
```

6. **Add box as a ramp:** Add a Cube as a ramp in front of the characters:

7. **Tag the Main Camera** game object with the tag *MainCamera*.

8. **Play!** Our character should be able to walk on the ramp and fall down again:

2.5.1.2 Extending the Controller with Jump Motion

Our controller can now walk up the ramp and it also falls back down again. But it falls with the same walk animation which is not realistic. We want to extent our controller with some jump and a crouch animation.

1. **Duplicate the Sintel Character:**

   a. **Duplicate the last character** and rename it *SintelWalkRunAndJumpControlled*.

   b. **Duplicate its animator controller** and rename it *WalkRunAndJumpAnimator* and assign it to the character.

   c. **Duplicate the script** *PlayerWalkAndRunBehaviour* and rename it *PlayerWalkRunAndJumpBehaviour*.

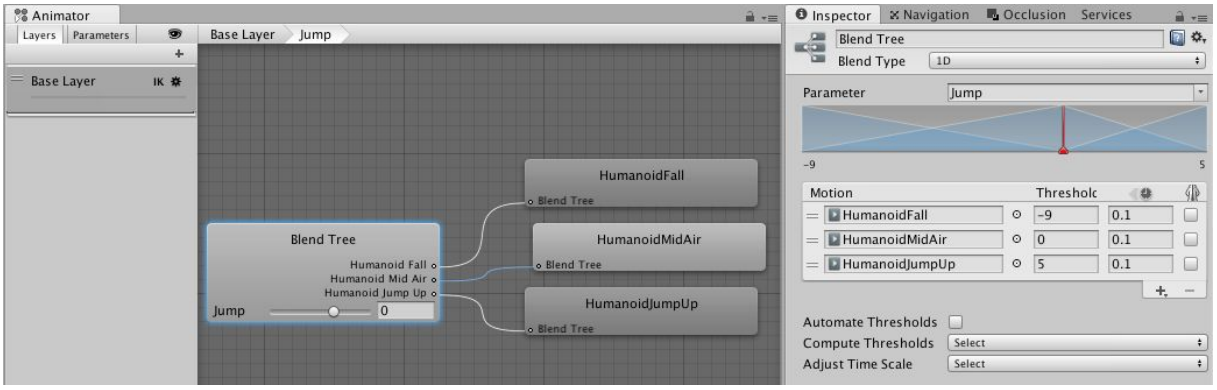
2. **New Blend Tree for Jump Animation**:

   a. Right-click on the Animator window work area and choose *Create State > From New Blend Tree*.
   - Name it *Jump* in the Inspector.
   - Leave *Foot IK* unchecked.

   b. Open the blend tree by double click and set the blend type to *1D*.

   c. Add in the *Parameter* tab a new float parameter and name it *Jump*.

   d. Add 3 *Motion Field* lines and insert the 3 animations *HumanoidFall*, *HumanoidMidAir* and *HumanoidJumpUp* with a the according threshold values -9, 0 and 5:



If you analyze the 3 animations you will see that they are only two frames long each. Our full jump animation is therefore nothing else than a blend between 3 frames.

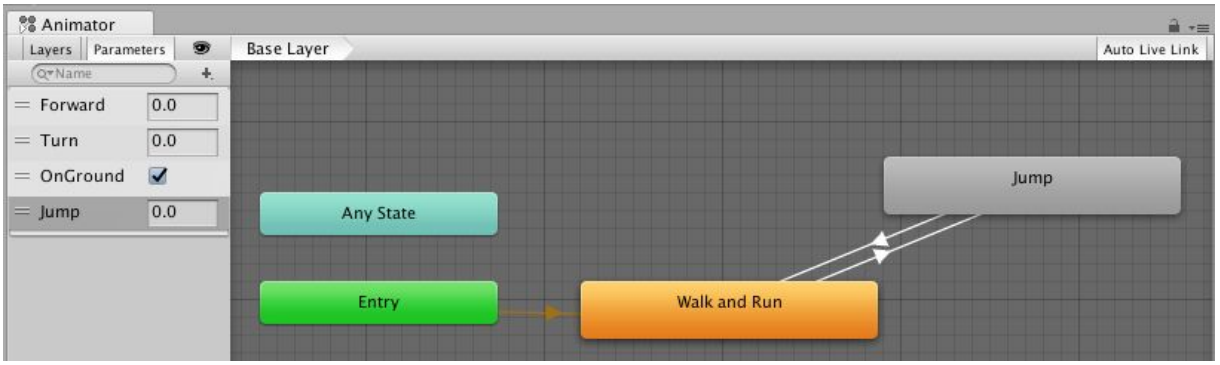
3. **Checking the Ground Status**: The ramp up walking and falling back to the ground is done automatically by the rigidbody and the capsule collider component. If we want to blend over to a jump animation when falling down, we have to check whether we are on the ground or not. We do this in a new routine *GetGroundStatus* by shooting a raw downwards with a specific length *groundCheckDistance* of initially 0.3:

```
...
public float        groundCheckDistance = 0.3f;
...

bool GetGroundStatus()
{
    RaycastHit hitInfo;

    #if UNITY_EDITOR
    // helper to visualise the ground check ray in the scene view
    Debug.DrawLine(transform.position + (Vector3.up * 0.1f),
                   transform.position + (Vector3.up * 0.1f) +
                   (Vector3.down * groundCheckDistance), Color.red);
    #endif

    // Shoot ray downwards to check ground state
    return Physics.Raycast(transform.position + (Vector3.up * 0.1f),
                           Vector3.down,
                           out hitInfo,
                           groundCheckDistance);
}
```

4. **Add new Animator Parameters**: We also want to actively jump when we press the spacebar. We therefore add a boolean parameter *OnGround* to the Animator.

5. **Add State Transitions** between the *Walk and Run* and *Jump* state:
   - From *Walk and Run* to *Jump* set the condition *OnGround* = false.
   - From *Jump* to *Walk and Run* set the condition *OnGround* = true.

- Uncheck in both transitions *Has Exit Time*.



6. **Get Jump Input in FixedUpdate**: To actively jump we get another input for the spacebar and pass it to the *Move* method.

```
private void FixedUpdate()
{
    // Read inputs
    float inputH = Input.GetAxis("Horizontal");
    float inputV = Input.GetAxis("Vertical");
    bool  doJump = Input.GetButtonDown("Jump");

    // Calculate move direction from the camera orientation
    _moveDir = inputV*_cam.forward + inputH*_cam.right;

    // We scale down the move vector for walking
    if (Input.GetKey(KeyCode.LeftShift)) _moveDir *= 0.5f;

    Move(_moveDir, doJump);
}
```

7. **Add some Variables**:

```
public class PlayerWalkRunAndJumpBehaviour: MonoBehaviour
{   ...
    public float        groundCheckDistance = 0.3f;
    public float        jumpPower = 6.0f;
    ...
    private bool        _isGrounded;
    private float       _origGroundCheckDistance;


    private void Start()
    {   ...
        _origGroundCheckDistance = groundCheckDistance;
    }
```

8. **Extend the Move method**:

   a. If we are on the ground and user pressed jump and the animator is in the *Walk and Run* state we apply a jump power to the upward component and leave the xy-components of the rigidbody's velocity.
   In the else case we are in the air and we reduce the *groundCheckDistance* if we are upwards going to avoid the ground intersection.

   b. To the animator we pass always the ground status *_isGrounded* and only if we are in the air we set the *Jump* parameter to the vertical velocity to blend the jump animation:

```
private void Move(Vector3 moveDir, bool doJump)
```

```
{   ...
    transform.Rotate(0, _turnAmount * turnSpeed * Time.deltaTime, 0);

    _isGrounded = GetGroundStatus();

    if (_isGrounded)
    {   if (doJump &&
            _animator.GetCurrentAnimatorStateInfo(0).IsName("Walk and Run"))
        {
            _rigidbody.velocity = new Vector3(_rigidbody.velocity.x,
                                              jumpPower,
                                              _rigidbody.velocity.z);
            groundCheckDistance = 0.1f;
        }
    } else
    {   groundCheckDistance = _rigidbody.velocity.y < 0 ?
                                  _origGroundCheckDistance : 0.01f;
    }

    // update the animator parameters
    _animator.SetFloat("Forward", _forwardAmount, 0.1f, Time.deltaTime);
    _animator.SetFloat("Turn", _turnAmount, 0.1f, Time.deltaTime);
    _animator.SetBool("OnGround", _isGrounded);
    if (!_isGrounded)
        _animator.SetFloat("Jump", _rigidbody.velocity.y);
}
```

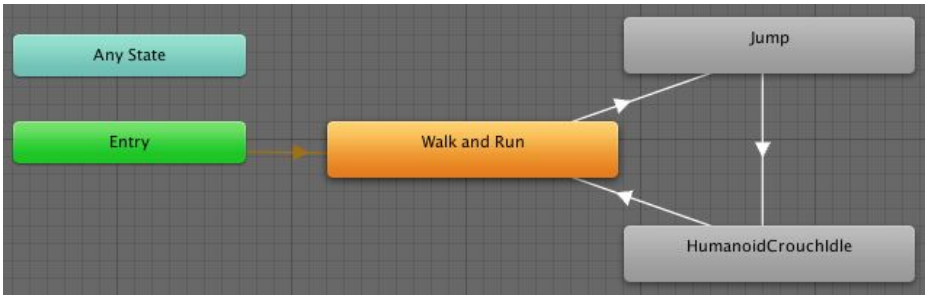
9. **Restrict Animator Move to Grounded**: We have to restrict now the velocity setting by the animator to the grounded case because the animator has no information over the horizontal motion during a jump. We already applied the jump power in *Move*.

```
public void OnAnimatorMove()
{
    if (_isGrounded && Time.deltaTime > 0)
    {
        // On ground we calculate the speed from the root motion
        Vector3 v = _animator.deltaPosition / Time.deltaTime;

        // We preserve the existing vertical part of the current velocity
        v.y = _rigidbody.velocity.y;

        _rigidbody.velocity = v;
    }
}
```

10. **Play!** The Sintel character should now be able to jump from standing, from running and when falling. You can set the jump height with the Jump Power variable.

References: This tutorial is based on Unity's *ThirdPersonController* that you can find in *Standard Assets/Characters/ThirdPersonController/Prefabs*.

Checkpoint 5: Walk, Run and Jump Controller: You can make this character controller even more realistic by adding a crouch animation between the *Jump* and *Walk and Run* state:

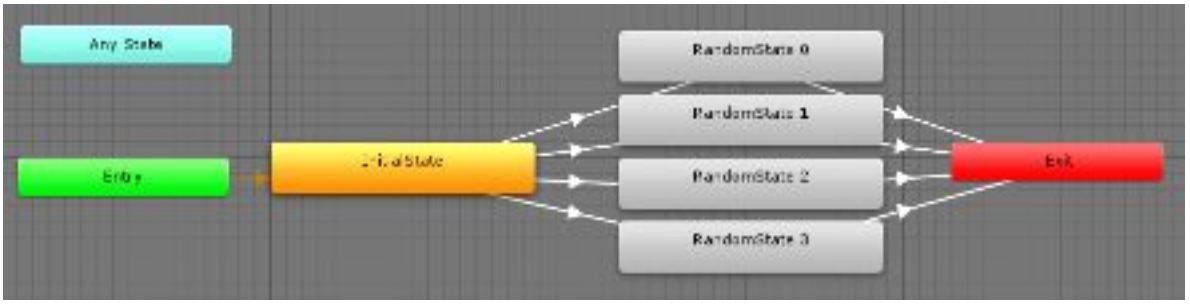


2.5.2 State Machine Behaviours

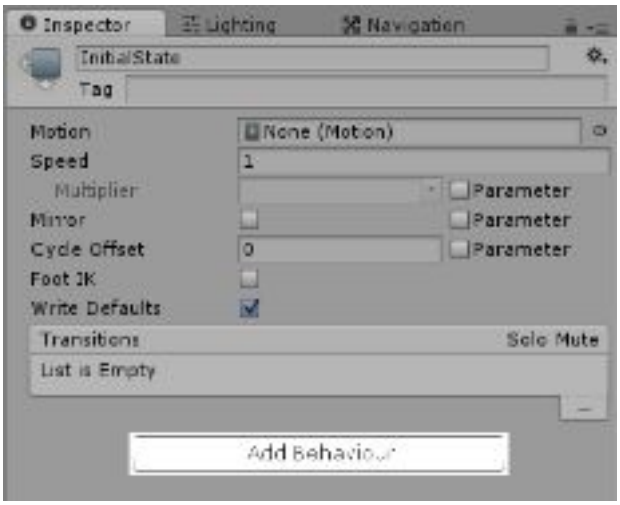
The *State Machine Behaviour* is a script that can be attached to animator states. This is similar to how *MonoBehaviours* work for *Game Objects*. These classes can be used to add all sorts of behaviour that is state dependent such as playing sounds whenever you enter a state. They can even be used to make logic state machines which are independent of animation.

Here is a quick example on how to use State Machine Behaviours. Say we want an Animator Controller that plays a random animation, these are the steps to take:

1. **Create a new Animator Controller.**
2. **Create a new empty state:** Call this state InitialState. Add any animation to this state, this state will be the one from where we will choose a random next animation.
3. **Create as many extra empty states as you'd like:** Create extra states and fill them with random animations that you'd like to play after the state created in step 2.
4. **Add a new int parameter and call it RandomIndex.**
5. **Create transitions from our InitialState to all our other states:** As the single transition condition add "RandomIndex Equals X" to each, where X is numbered 0 to n-1 and n is the total amount of transitions.



6. **Add a State Machine Behaviour** to our *InitialState* by clicking on Add Behaviour in the Inspector while having it selected. You will be presented with a script that has all of the possible callbacks added and commented out.




```
public class RandomAnimSMB : StateMachineBehaviour
{
    // The number of random states to choose between.
    public int numberOfStates = 4;

    // For referencing the RandomIndex animator parameter.
    private readonly int _hashRandomIndexPara =
                        Animator.StringToHash("RandomIndex");

    // OnStateEnter is called when a transition starts and the state machine
    // starts to evaluate this state
    override public void OnStateEnter(Animator animator,
                                      AnimatorStateInfo stateInfo,
                                      int layerIndex)
    {   // Set Random Index based on how many states there are.
        int randomSelection = Random.Range(0, numberOfStates);
        animator.SetInteger(_hashRandomIndexPara, randomSelection);
    }
}
```

Note: There is a difference between a State Machine Behaviours for a Sub State Machine and a normal state. Please refer to the comments when generating a new Sub State Machine for the specific case.

As an additional example you can look at the Unity Labs project to see how they used *State Machine Behavior* to implement the entire character handling.
https://www.assetstore.unity3d.com/en/#!/content/33835

For additional information you can visit Unity's tutorial about *State Machine Behaviours*:
https://unity3d.com/learn/tutorials/modules/beginner/5-pre-order-beta/state-machine-behaviours

2.6 Unity Blend Shapes

Unity's Blend Shapes or ***Blenders Shape Keys*** or ***Mayas Morph Targets*** or ***per-vertex animations*** or ***shape interpolations*** are alle synonyms for an alternative method of 3D shape animation instead of using skeletal animation.

For each blend shape a separate set of vertices with different positions is stored. At run time the final position of each vertex is then calculated by a *linear interpolation* (aka LERP) between the initial position and the position of the vertex from the blend shape.

Blend shapes are often used for *facial animations* where we can interpolate between different *facial expressions*:

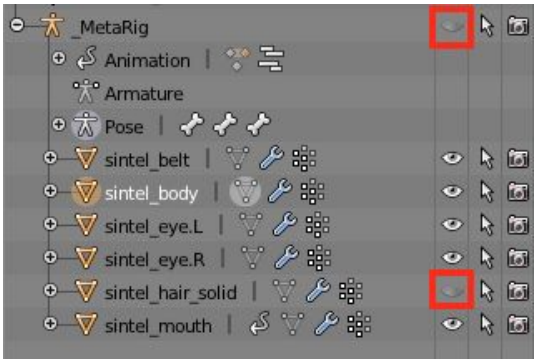
The **advantage of blend shapes** over skeletal animation is that the artist has more freedom over fine details of an expression. In the face the shape is largely influenced by muscles, more than by the two bones, the skull and the jaw. There are rigs for the the face but they are difficult to build and pose. On the other hand is a full body animation mainly influenced by the skeleton why it makes sense to use a skeleton rig to deform it.

Performance Considerations: Blend shape calculate their final vertex position at runtime once per frame as a mix of the different triangle mesh version. This is similar to skeleton animation where each vertex gets its position as a sum of vertex weights of the different influencing bones. This loop over all vertices has to be done either on CPU or on the GPU. A disadvantage of blend shape might be its higher memory footprint. The rendering system has to hold a copy of the entire mesh for each blend shape. If we apply blend shapes only to the face I can imagine that it is advantageous to split the head in a separate mesh with a lot less vertices than the full body. On the other hand does each mesh object cause a draw call.

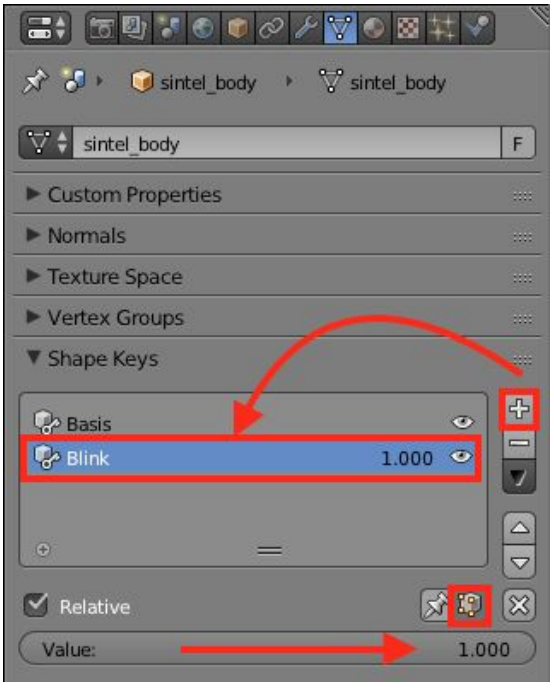
2.6.1 Adding Shape Keys in Blender

1. **Duplicate your last Unity Project** and rename the copy to *2.6 Shape Keys*.
2. **Delete all characters except the last Sintel character** with the fully functional character controller.
3. **Set a static camera in front of the face.**
4. **Prepare Blender**:
   - Open the *Sintel* character in Blender by double clicking it in Unity.
   - In *Object Mode* set the orthographic front view with [1].

- Hide the *_MetaRig* and the *sintel_hair_solid* object in the outliner view:



5. **Create new Shape Keys**:
   - Select the *sintel_body* object.
   - Press two times the plus button in the *Shape Keys* area of the *Properties* view.
   - Rename the second shape key to *Blink.*
   - Select the option that the shape keys are applied in *edit mode*.
   - Slide the *Value* slider to the right and the value 1.0.

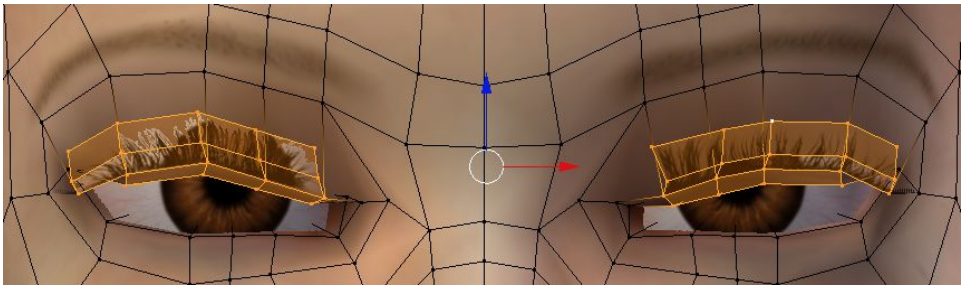


6. **Edit the *Blink* Shape Key**: For making good shape keys it is essential to have good reference images or even better a mirror where you can look at yourself. Another possibility is to use your cell phone or tablet with the selfie camera turned on.

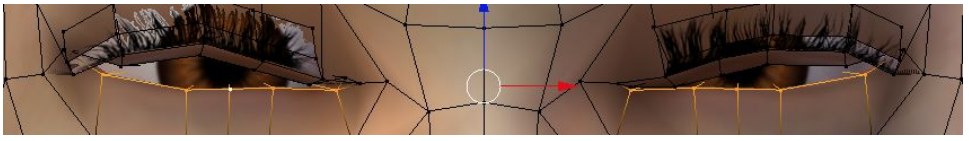
   a. In the *Edit* mode make sure you have disabled the backface vertices for not selecting accidentally vertices on the back of the head:

   b. Make sure the *Blink* shape key is still selected and the value slider is on 1.0!

c. Select the top vertices of the eye and eyelash with the circle select tool [C] and move them down about ⅔ of the eye with the blue arrow:



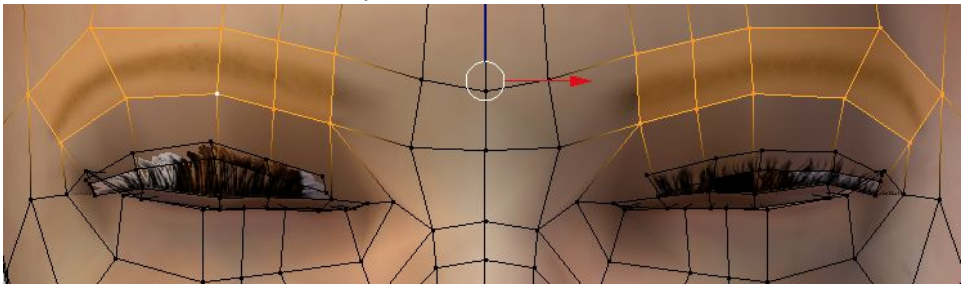
d. Select the lower vertices of the eye and move them up:



e. Close the eyes by moving the remaining vertices one by one.
Check also the depth from the side view [3].

f. Select the upper vertices of the eyelash and bring them down a bit.

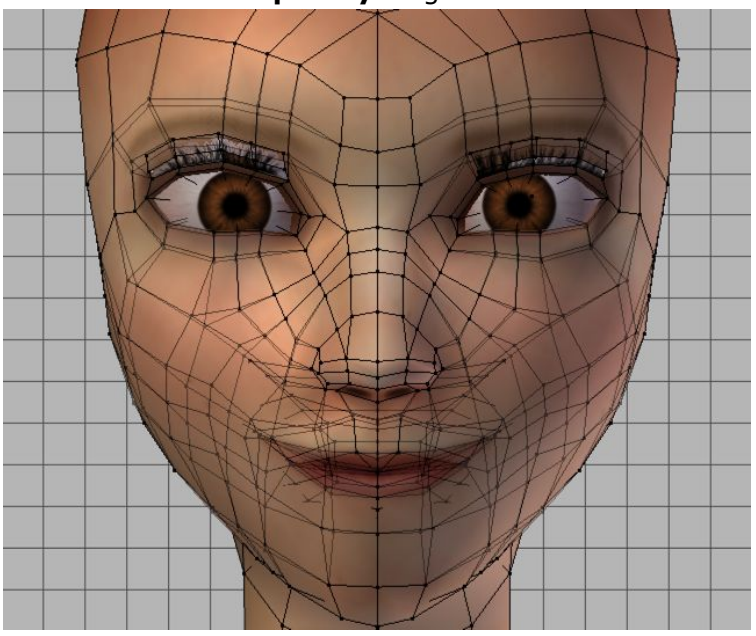
g. Select the vertices of the eyebrows and lower them also a bit:



h. Now the test the shape key by sliding the value slider between 1 and 0!

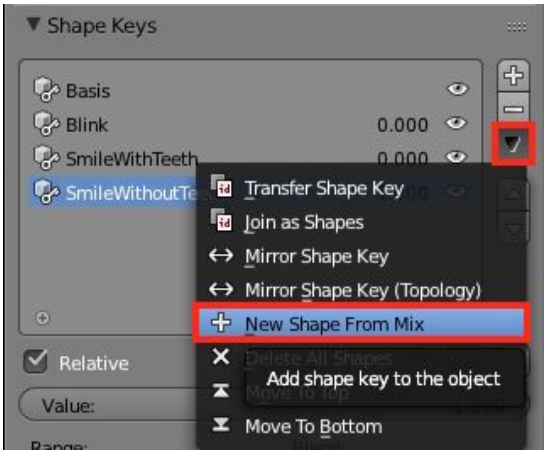
7. Create a *Smile* Shape Key:
- Create another shape key by pressing the plus button.
- Rename it to *Smile*.
- Slide its value slider to 1
- Slide the value slider of the *Blink* shape key to 0

8. **Edit the *Smile* Shape Key**: A good smile influences almost the entire face:

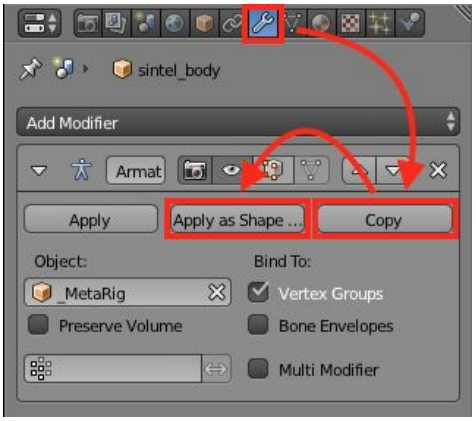


9. **Reset all shape keys and save the Blender file** and switch to Unity.

10. **New Shapes from Mix**: An alternative way to create a new blend shape is to create a copy of the current mix of existing blend shapes.



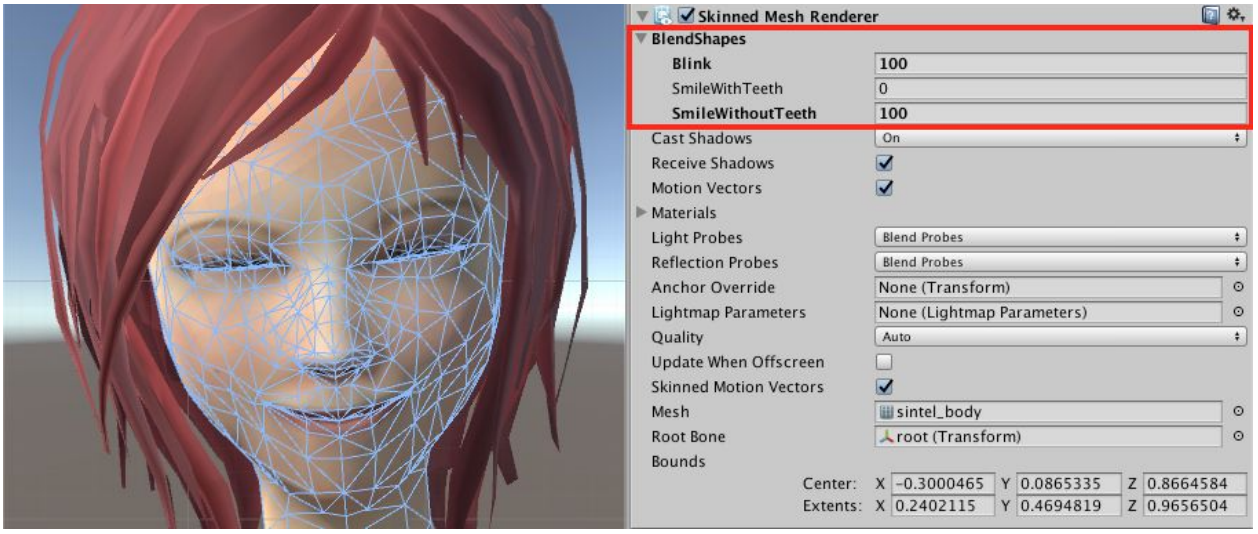
11. **Shape Key from Pose**: Another alternative is to create a shape key from skeleton pose. If our model has teeth and a jaw bone as a child of the head bone we can open the mouth in *Pose Mode*.
    - Select the mesh that is deformed by the armature and go into *Object Mode*.
    - In the *Armature Modifier* click *Copy*
    - In the *Armature Modifier* click *Apply as Shape …*
    - In *Pose Mode* reset all poses with *Pose > Clear Transform > All*
    - Select the mesh and you will see a new blend shape in the *Blend Shapes* section.
    - This method works even in combination with active blend shapes.



2.6.2 Controlling Blend Shapes in Unity

In Unity the *Blenders shape keys* appear as *Blend Shapes* in a new section of the *Skinned Mesh Renderer* component of the selected object that contains the shape keys.

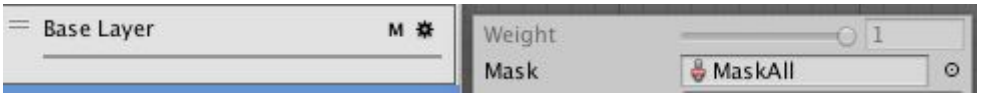
You can adjust each blend shapes value between 0 and 100 by hand, by script or by animation:



We will apply the two shape keys in two additional animator layers.

- In the first layer we will let blink the eye constantly once every 5 seconds and slightly move the smile key.
- In the second layer we will reimplement our waving gesture and additionally smile.

1. **Layer for Blend Shapes**: In our *WalkRunAndJumpAnimator* add a second layer and name it *Blend Shapes*.
   - Set its weight to 1
   - Add an avatar mask where all body all parts are masked.



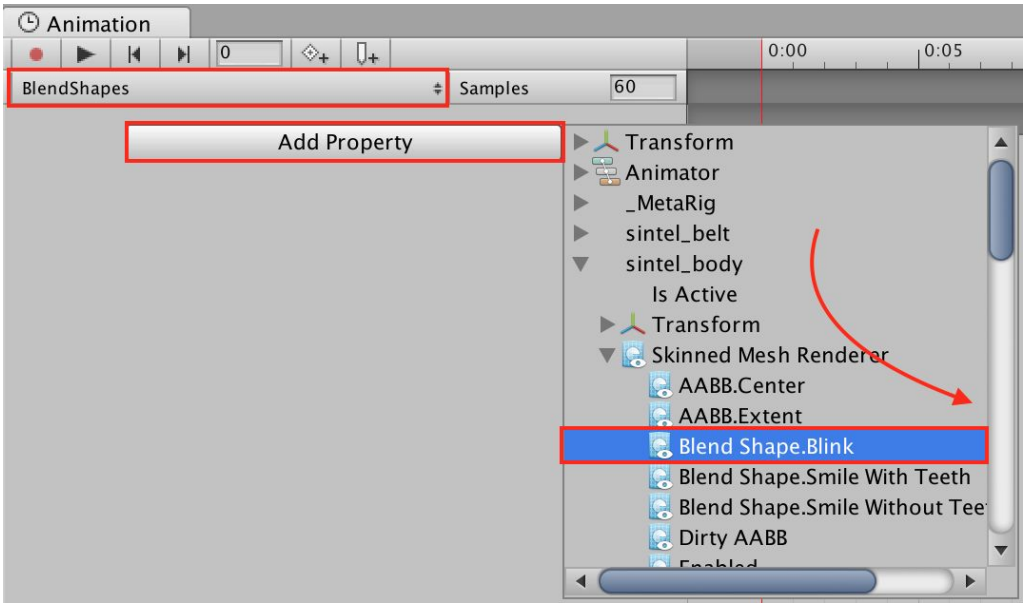
2. **Create new Animation for Blend Shapes**: In the *Animation* folder create a new *Animation* and name it *BlendShapes*.

3. **Drag the animation into the Blend Shape layer** of the animator window.

4. **Open the Animation Window** with *Window* > *Animation* and select the *sintel_body* object. You will see all animation listed.

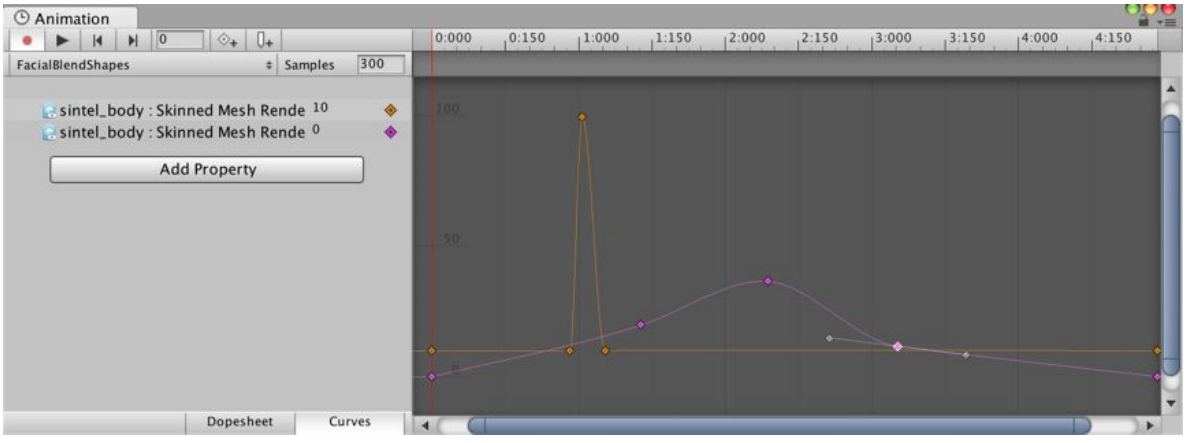
5. **Select the new BlendShape animation**

6. **Add Blink Property** by clicking *Add Property* which opens a hierarchy
   - Select our *Blend Shape.Blink* property
   - Scroll to the right and click the plus sign to add it.



7. **Add Smile Property** in the same way.

8. **Add key points to define the blink and a slight smile animation**: The goal is that Sintel blinks once every 5 seconds the face gets some slight motion just to see that the face is alive:



9. **Drag the *BlendShapes* animation into the *Blend Shapes* layer** of the animator:

10. **Play!**: Sintel should blink now.

11. **Add a third Layer for the *Waving* animation**.

12. In our ***PlayerWalkRunAndJumpBehaviour*** script we add the *Waving* when we press the Tab key. In the new method *Wave* we activate the smiling by setting the blend shape weight to 100:

```
public float                waveLiftSpeed = 4.0f;
public GameObject           SintelBody;
private SkinnedMeshRenderer _meshRenderer;
private float               _waveWeight = 0.0f;

private void Start()
{   ...
    _meshRenderer = SintelBody.GetComponent<SkinnedMeshRenderer> ();
}

private void FixedUpdate()
{   ...
    bool  doWave = Input.GetKey("tab");
    ...
    Move(_moveDir, doJump);
    Wave(doWave);
}

private void Wave(bool doWave)
{
    if (doWave)
    {   // Lerp in the wave weight
        _waveWeight = Mathf.Lerp(_waveWeight, 1.0f, Time.deltaTime * waveLiftSpeed);
        _animator.SetLayerWeight(2, _waveWeight);
        _meshRenderer.SetBlendShapeWeight(1, _waveWeight * 100);
    }
    else
    {
        if (_waveWeight > 0.01f)
        {   // Lerp out the wave weight
            _waveWeight = Mathf.Lerp(_waveWeight, 0.0f, Time.deltaTime * waveLiftSpeed);
            _animator.SetLayerWeight(2, _waveWeight);
            _meshRenderer.SetBlendShapeWeight(1, _waveWeight * 100);
        }
    }
}
```

Checkpoint 6: Blend Shapes: Give a smile!

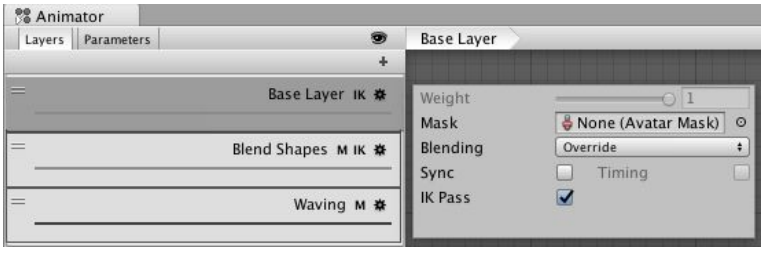
2.7 Unity Inverse Kinematics

We have already seen how useful inverse kinematics are to animators. Many games today use IK to achieve procedural animations. In most cases IK is used to correct existing animations such as correctly picking up an object. Some games even completely do without animations and simulate the movement of characters based on a physical model. Mecanim comes with some basic IK functionality built in, in this small chapter we will look at a few use cases of that functionality.

To set up IK for a character, you would have objects around the scene that the character interacts with, and then set up the IK in a script by dedicated Animator methods in the *OnAnimatorIK* callback method:

- *SetIKPositionWeight* & *SetIKPosition* control the positions of hands and feet.
- *SetIKRotationWeight* & *SetIKRotation* control the rotation of hands and feet.
- *SetIKHintPositionWeight* & *SetIKHintPosition* control the pol targets for the knees and elbows.
- *SetLookAtWeight* & *SetLookAtPosition* control the rotations of the body, head and eyes to make the character look at the given point.

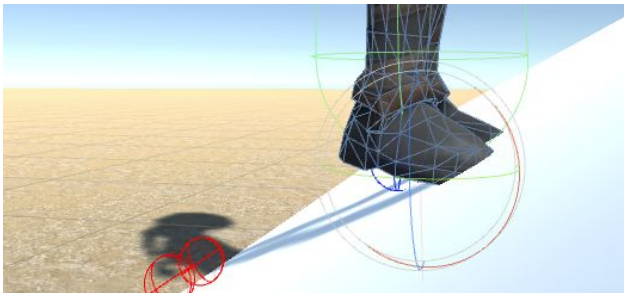
To enable the IK callback we have the activate the *IK Pass* in the animator layer:



Most parts in this chapter were inspired by the video tutorial The Build-In IK System.

2.7.1 Improve Standing with Foot IK

Even if we have enabled *Foot IK* in the properties of our *Walk and Run* animator the feet are not perfectly still on the ground. Another disadvantage is that the underground's normal is not evaluated and the feet is placed always horizontal:



We can improve this by checking the ground normal ourself with a raycast.

1. **Duplicate your last Unity Project** and rename the copy to *2.7 IK*.

2. **Create another C#-script component** on the Sintel character, name it *IKBehaviour* and paste the following code:

    a. In every frame we call in *Update* the method *GetFeetHits* where we do a raycast for both feet. We do this only when the character is standing still. If we hit the ground we calculate the hit point and the rotation at the hit point between the hit normal

and the vertical axis. Both the hit position and the hit rotation can be corrected with an additional offset. The total foot rotation is the sum of the ground rotation the additional offset rotation and the characters rotation and is calculated by multiplying its quaternions.

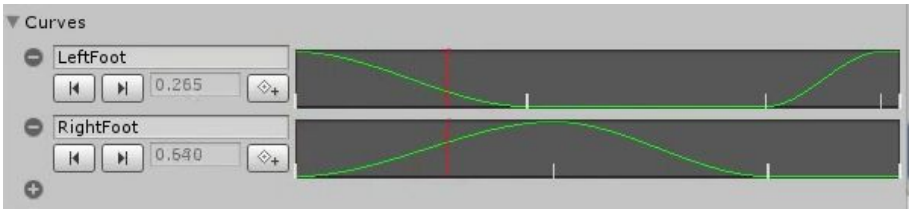
b. In the IK callback *OnAnimatorIK* we call *DoFootIK* where we apply the weight, the foot positions and rotations with the animator SetIK-methods.

```
public class IKBehaviour : MonoBehaviour
{   private Animator    _animator;

    public  bool        doFootIK = true;
    public  float       feetOffsetY = 0.12f;
    public  Vector3     footLeftOffsetEulerAngles = new Vector3(-6,-5,0);
    public  Vector3     footRightOffsetEulerAngles = new Vector3(-6, 8,0);

    private Transform   _footL;
    private Transform   _footR;
    private Vector3     _footL_HitPos;
    private Vector3     _footR_HitPos;
    private Quaternion  _footL_HitRot;
    private Quaternion  _footR_HitRot;
    private float       _feetIKWeight = 0.0f;

    void Start ()
    {
        _animator = GetComponent<Animator>();

        // Get the feet transforms from the animator
        _footL = _animator.GetBoneTransform(HumanBodyBones.LeftFoot);
        _footR = _animator.GetBoneTransform(HumanBodyBones.RightFoot);

        // Initialize the hit rotation so they are not null
        _footL_HitRot = _footL.rotation;
        _footR_HitRot = _footR.rotation;
    }

    void Update () {GetFeetHits();}

    void OnAnimatorIK() {DoFootIK();}

    void GetFeetHits()
    {
        // get animator variables
        float forward = _animator.GetFloat("Forward");
        float turn = _animator.GetFloat("Turn");
        bool  isGrounded = _animator.GetBool("OnGround");

        // Only do foot IK standing still
        _feetIKWeight = doFootIK && (forward+turn)<0.05f && isGrounded ? 1.0f : 0.0f;

        if (_feetIKWeight == 1.0)
        {
            // Do a ray cast for the left foot in world space downwards
            RaycastHit hit;
            Vector3 posWS = _footL.TransformPoint(Vector3.zero);
            if (Physics.Raycast(posWS, -Vector3.up, out hit, 1))
            {   _footL_HitPos = hit.point + new Vector3(0,feetOffsetY,0);
                _footL_HitRot = Quaternion.FromToRotation(transform.up, hit.normal) *
                                Quaternion.Euler(footLeftOffsetEulerAngles) *
                                transform.rotation;
            }

            // Do a ray cast for the right foot in world space downwards
            posWS = _footR.TransformPoint(Vector3.zero);
            if (Physics.Raycast(posWS, -Vector3.up, out hit, 1))
            {   _footR_HitPos = hit.point + new Vector3(0,feetOffsetY,0);
```

```
                _footR_HitRot = Quaternion.FromToRotation(transform.up, hit.normal) *
                                Quaternion.Euler(footRightOffsetEulerAngles) *
                                transform.rotation;
            }
        }
    }

    void DoFootIK()
    {
        _animator.SetIKPositionWeight(AvatarIKGoal.LeftFoot, _feetIKWeight);
        _animator.SetIKPositionWeight(AvatarIKGoal.RightFoot, _feetIKWeight);
        _animator.SetIKPosition(AvatarIKGoal.LeftFoot, _footL_HitPos);
        _animator.SetIKPosition(AvatarIKGoal.RightFoot, _footR_HitPos);

        _animator.SetIKRotationWeight(AvatarIKGoal.LeftFoot, _feetIKWeight);
        _animator.SetIKRotationWeight(AvatarIKGoal.RightFoot, _feetIKWeight);
        _animator.SetIKRotation(AvatarIKGoal.LeftFoot, _footL_HitRot);
        _animator.SetIKRotation(AvatarIKGoal.RightFoot, _footR_HitRot);
    }
}
```

3. **Adjust Foot Offset Values**: Because the raycast hit position is not equal to the foot bone we have to adjust the position and the orientation with variables *feetOffsetY*, *footLeftOffsetEulerAngles* and *footRightOffsetEulerAngles*:

4. **Apply Foot IK during Walking**: In the video tutorial The Built-In IK System the author proposes to use the Foot IK beside the idle animation also during the walk animation. He lets the IK weight of each foot be controlled by a two additional parameters *LeftFoot* and *RightFoot*. The variables values are set by curves that can be defined in the animations settings. When the feet are flat on the ground the values should be 1 otherwise 0:



2.7.2 Make a character look at an object using Look IK

Another application for IK is to let a character look at a given position. Amount of body, head and eye rotation is controlled with a separate weight for each of them. To avoid unnatural bending and rotation you have to set a clamp value > 0.

Make also sure that you enable the *IK Pass* in the *Blend Shapes* layer (layer Nr. 1). The layer mask would otherwise block any rotation of the head.

1. **Add the following parts to the IKBehaviour Script**:

```
public class IKBehaviour : MonoBehaviour
{   ...
    public  Transform   lookAtPoint;
    public  float       lookIKWeight = 0.0f;
    public  float       lookIKBodyWeight = 0.3f;
    public  float       lookIKHeadWeight = 0.9f;
    public  float       lookIKEyesWeight = 1.0f;
    public  float       lookIKClampWeight = 0.5f;
    ...

    void OnAnimatorIK()
    {   DoFootIK();
        DoLookIK();
    }

    void DoLookIK()
    {   _animator.SetLookAtWeight(lookIKWeight,
                                  lookIKBodyWeight,
                                  lookIKHeadWeight,
                                  lookIKEyesWeight,
                                  lookIKClampWeight);
        _animator.SetLookAtPosition(lookAtPoint.position);
    }
}
```

2. **Add a Sphere Game Object** and place it in front of Sintels head and scale it down to 0.1. Assign the sphere object as the public *lookAtPoint* variable.

3. **Adjust the look IK weights** for the body, head and the eyes:

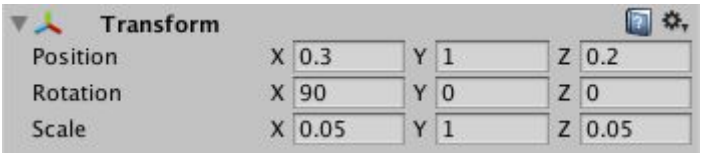
2.7.3 Making a character hold a static object using Hand IK

Let's make Sintel hold to a cylinder object with his right hand.

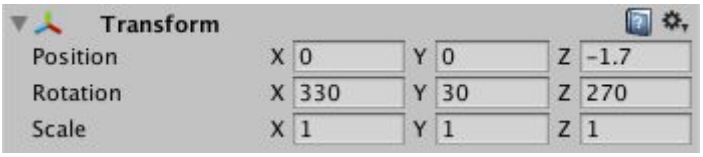
1. **Add a Cylinder and Handle Object**:

   a. Open the project and make sure that Sintel is at position [0,0,0] and has the scale [1,1,1].

b. Add a cylinder with *Game Object > 3D Object > Cylinder*. Set its transform as follows:



c. Select in the Hierarchy window the *cylinder* and add an empty object by *right-click > Create Empty*. Name it *Handle*.
Transform the empty object as follows:



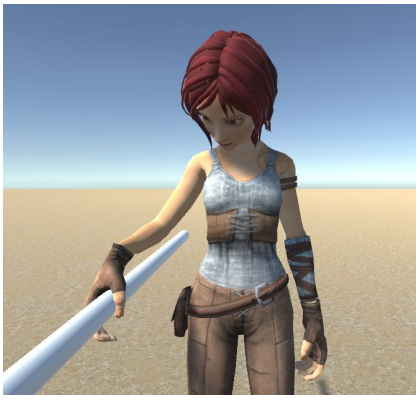
2. **Add the following parts to the IKBehaviour Script**: We initiate the cylinder grabbing with the key G in *FixedUpdate*. The weight and position setting is done in the IK callback *OnAnimatorIK*:

```
public class IKBehaviour : MonoBehaviour
{   ...
    public  bool        handIsHolding = false;
    public  Transform   handRightHandle;
    public  float       handGrabSpeed = 2.0f;

    private bool        _handDoGrabOrRelease;
    private float       _handGrabWeight = 0.0f;
    private float       _handKeyDownTime;
    ...

    void OnAnimatorIK()
    {   DoFootIK();
        DoLookIK();
        DoHandIK();
    }

    void DoHandIK()
    {   if (Input.GetKeyDown(KeyCode.G) && Time.time > _handKeyDownTime + 0.5f)
        {   handIsHolding = !handIsHolding;
            _handKeyDownTime = Time.time;   // keep time to avoid double firing events
        }

        if(handIsHolding && handRightHandle != null)
        {   // Lerp in the grab weight
            _handGrabWeight = Mathf.Lerp(_handGrabWeight, 1.0f,
                                         Time.deltaTime * handGrabSpeed);
            _animator.SetIKPositionWeight(AvatarIKGoal.RightHand, _handGrabWeight);
            _animator.SetIKRotationWeight(AvatarIKGoal.RightHand, _handGrabWeight);
            _animator.SetIKPosition(AvatarIKGoal.RightHand,handRightHandle.position);
            _animator.SetIKRotation(AvatarIKGoal.RightHand,handRightHandle.rotation);
        } else
        {   if (_handGrabWeight > 0.01f)
            {   // Lerp out the grap weight
                _handGrabWeight = Mathf.Lerp(_handGrabWeight, 0.0f,
                                             Time.deltaTime * handGrabSpeed);
                _animator.SetIKPositionWeight(AvatarIKGoal.RightHand, _handGrabWeight);
                _animator.SetIKRotationWeight(AvatarIKGoal.RightHand, _handGrabWeight);
                _animator.SetIKPosition(AvatarIKGoal.RightHand,handRightHandle.position);
                _animator.SetIKRotation(AvatarIKGoal.RightHand,handRightHandle.rotation);
            } else
            {   _animator.SetIKPositionWeight(AvatarIKGoal.RightHand, 0);
                _animator.SetIKRotationWeight(AvatarIKGoal.RightHand, 0);
```

```
            }
        }
    }
}
```

3. **Assign the *Handle*** object to the *HandRightHandle* variable.

4. **Play**: Press G to let the hand hold the bar:



2.7.4 Making a character carry an object

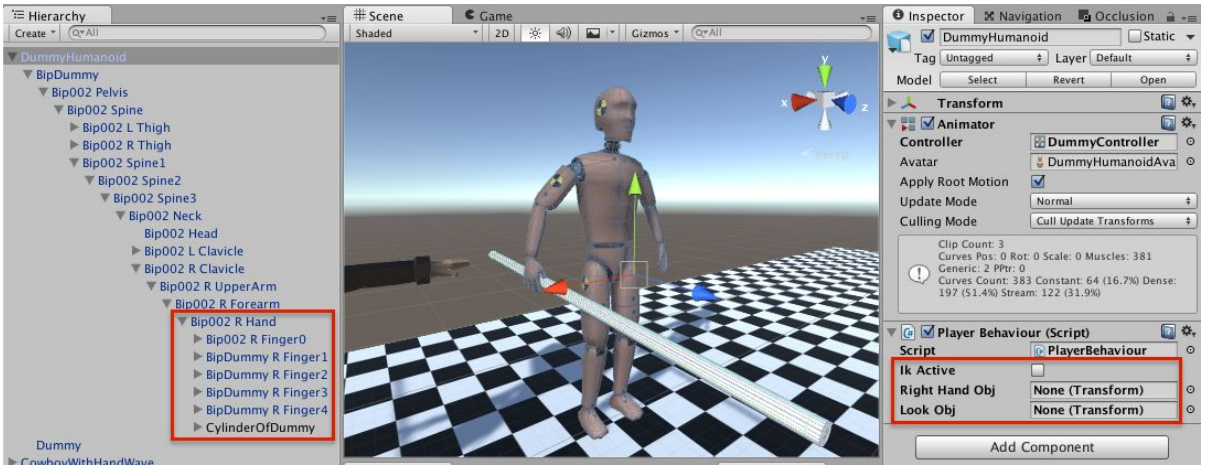
In the previous example we made our animated character hold onto a static object. The hand was following the object and not vice versa. This works fine as long as the character is in arm distance. If we want the character to carry the cylinder we have to make it a child of the character.

2.7.4.1 Making a character carry an object without IK

1. **Duplicate the character and the cylinder** with [cmd-D] and move it along the x-axis.

2. **Turn off the hand IK**.

3. **Make the cylinder it a child of the *hand* bone**:



4. **Play**: You will notice that the cylinder will do all the motions of the hand as if the cylinder has no weight.

2.7.4.2 Making a character carry an object with IK

1. Move the cylinder object up in the hierarchy to the spine bone. With this it will follow the much smoother motion of the spine bone.

2. Turn on again the hand IK:

3. **Play**: The hand will now still hold the cylinder with IK.

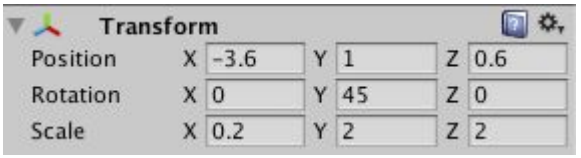
Checkpoint 7: IK: Extend the Hand IK part so that Sintel can grab the cylinder and can walk away with the cylinder in its hand. You have the make the cylinder a child to the hand or spline bone.

2.7.5 Making a character hit a wall with IK

As a final IK tutorial we will be looking at correcting limb positions in a running animation. For our example we will use a boxing animation.
The way this works is through animation curves. We embed a curve into the animation that holds the information of how much we should correct a given bone with IK. In case of our boxing animation this curve indicates the time the fist hits it's target.

1. **Add another Dummy character**: Place it at [-3,0,0] and name it *DummyBoxer*.

2. **Add a cube**: Name it *Wall* and transform it as follows:



3. **Add an empty object to the Wall**: Name it *HitTarget* and position it at [1, 0.2, 0].

4. **Create a new script:** Name it *BoxerBehaviour* and copy the following:

```
public class BoxerBehaviour : MonoBehaviour {
    private Animator _animator;
    public bool ikActive = false;
    public Transform rightHandObj = null;
    public Transform lookObj = null;
```

```
    void Start () {_animator = GetComponent<Animator>();}
    void Update () {}

    void OnAnimatorIK()
    {
        if (_animator)
        {   if (ikActive)
            {
                // Set the look target position, if one has been assigned
                if (lookObj != null)
                {    _animator.SetLookAtWeight(1);
                    _animator.SetLookAtPosition(lookObj.position);
                }

                // Set the right hand target position and rotation
                if (rightHandObj != null)
                {   float hitWeight = _animator.GetFloat("HitWeight");
                    _animator.SetIKPositionWeight(AvatarIKGoal.RightHand, hitWeight);
                    _animator.SetIKRotationWeight(AvatarIKGoal.RightHand, hitWeight*0.5f);
                    _animator.SetIKPosition(AvatarIKGoal.RightHand, rightHandObj.position);
                    _animator.SetIKRotation(AvatarIKGoal.RightHand, rightHandObj.rotation);
                }

            }
            // if the IK is not active, set the position and rotation of
            // the hand and head back to the original position
            else
            {   _animator.SetIKPositionWeight(AvatarIKGoal.RightHand, 0);
                _animator.SetIKRotationWeight(AvatarIKGoal.RightHand, 0);
                _animator.SetLookAtWeight(0);
            }
        }
    }
}
```

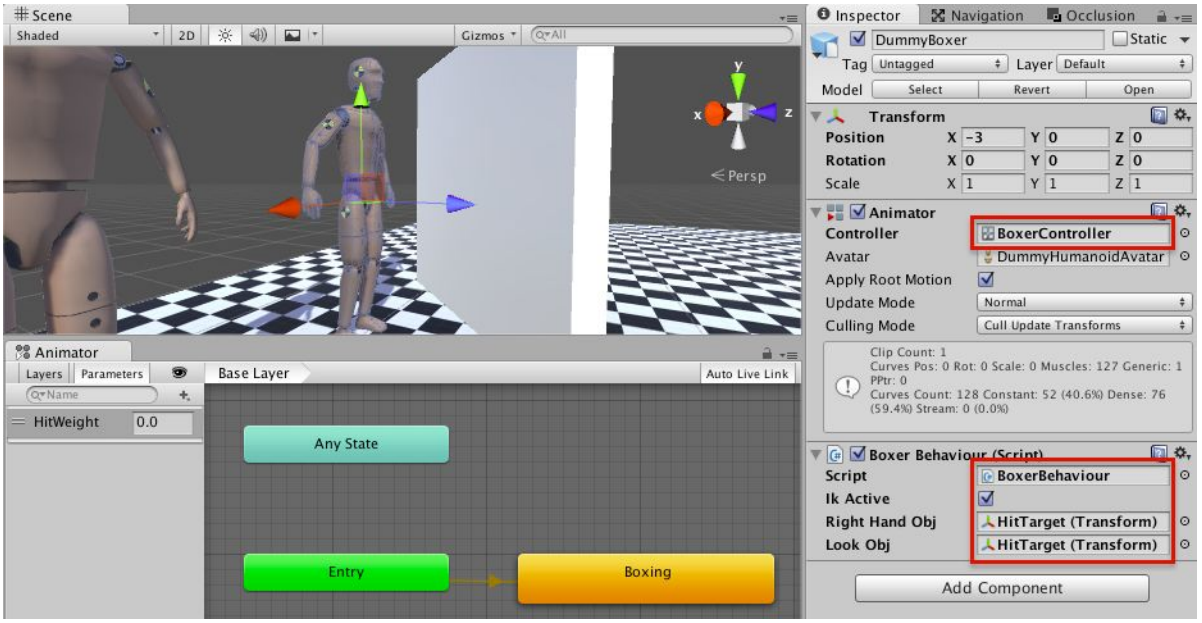
5. **Create a new character controller**: Name it *BoxerController*

    a. Add the *Boxing* animation.

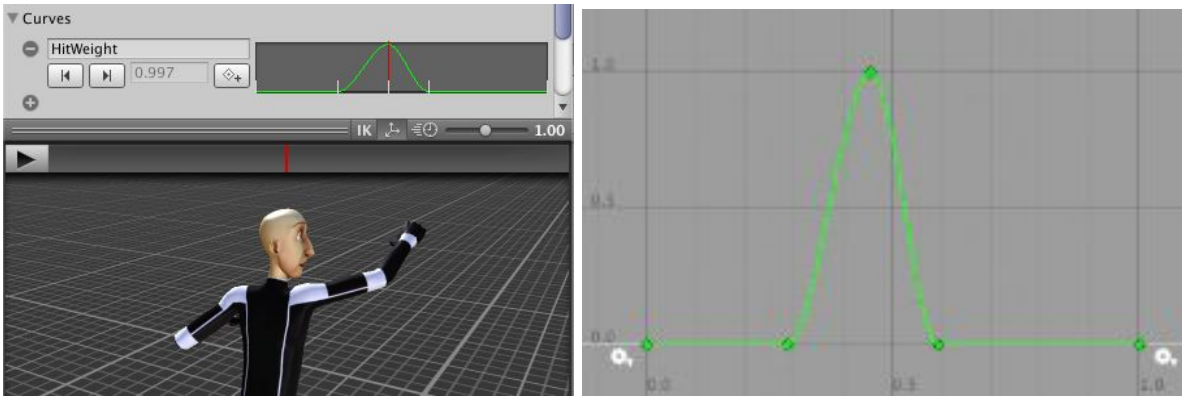
    b. Add the float parameter *HitWeight*.

    c. Turn on the layer's *IK Pass*.

6. **Assign everything**, the *BoxerController*, the *BoxerBehaviour* and the *HitTarget* to the *DummyBoxer*:



7. **Add a curve for the *HitWeight***: The boxer would box through the wall because the parameter *HitWeight* remains 0. Add a curve and name it *HitWeight*. Adjust the curve so that its max. value will be 1 at the max. extension of the arm:



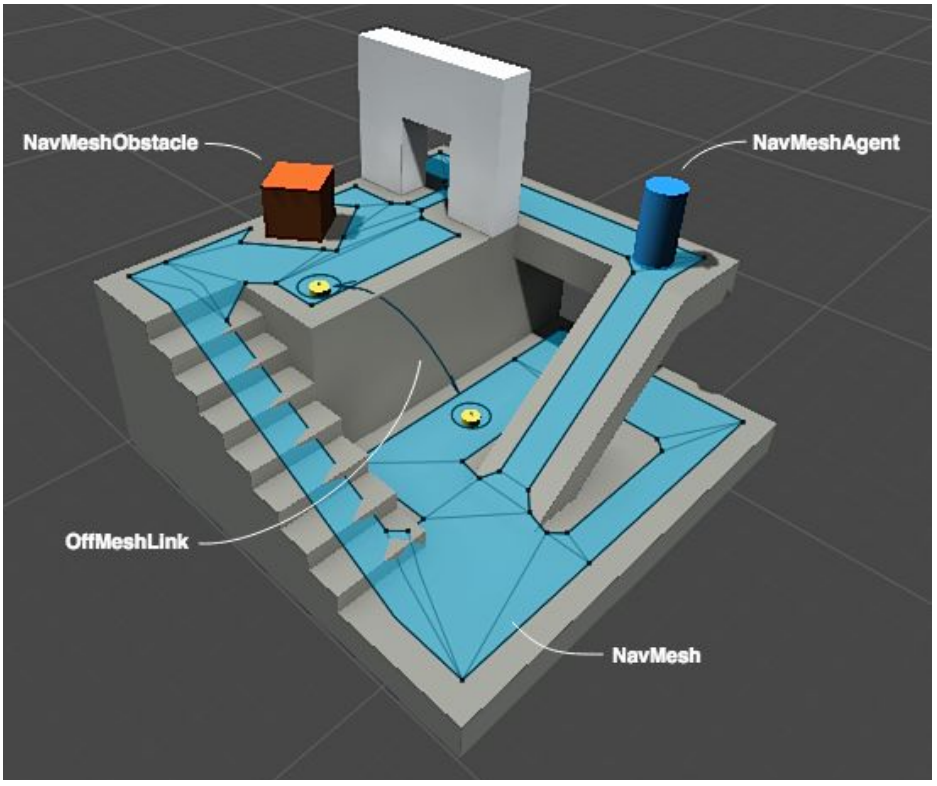
8. **Play!**

3 Navigation

In this chapter we will learn what the *Unity Navigation System* is and how we can use it. The Unity Navigation System is an ***Artificial Intelligence (AI)*** system for the ***pathfinding*** of computer controlled game objects. These objects are not controlled by the user input and are mostly called *agents* or *bots* (short for robots). The control of these objects is driven by sophisticated algorithms such as the A*-star pathfinding algorithm. You can learn more about Artificial Intelligence in the module *BTI7282 Virtual Reality & Artificial Intelligence*.

The Unity Navigation System contains the following parts:

- The **NavMesh** is the walkable navigation mesh where agents can pass. It can be generated or baked from a static environment.
- The **NavMesh Agent** component controls a game object. It lets the object find a target or following a path on the shortest possible path without colliding with obstacles or other agents.
- The **Off-Mesh Link** component allows an agent to jump from one position to another where there is no NavMesh underneath.
- The **NavMesh Obstacle** component can be attached to dynamic objects so that NavMesh agents bypass them.



3.1 Unity NavMesh

1. **Copy the last project** and rename it to *03 Navigation*.

   a. Remove everything but the *Sintel* character.

   b. Fix the scripts, so that they work errorless without IK goals.

c. Attach the *SmoothFollow* script that we used in the last semester to follow the car to a standard camera to follow our character. You can get the script from the Dropbox folder *03 Navigation*.

2. **Create a Scene:**

a. Copy the Dropbox folder *Shared Assets/Prototyping* into your *Assets* folder.

b. Add the the *FloorPrototype64x01x64* from the *Prototyping/Prefabs* folder as the floor a [0,0,0].

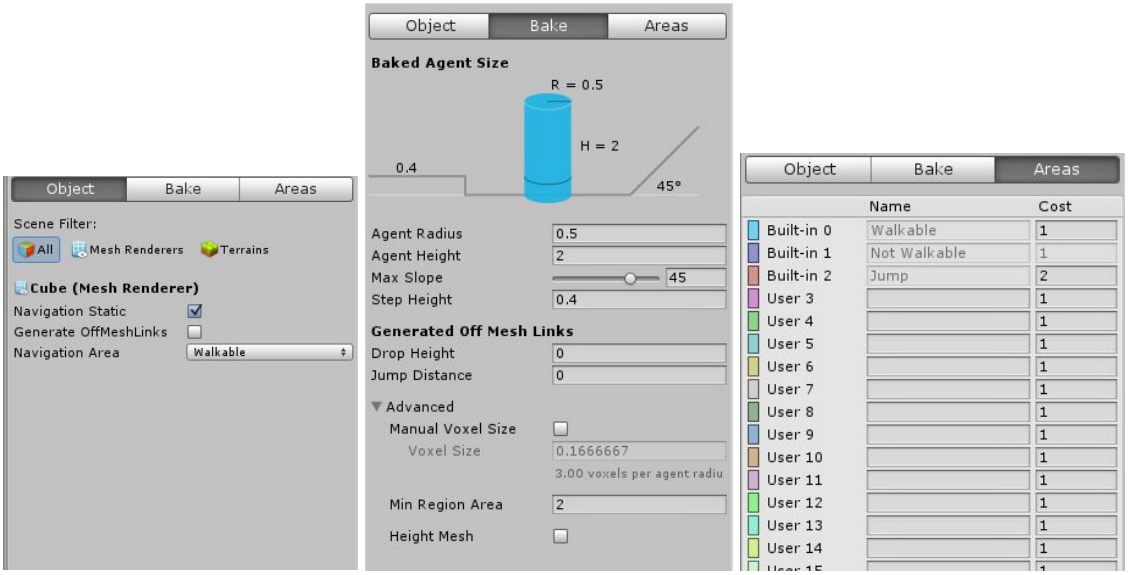
c. Add some cubes and ramps with different scales.

3. **Open up the Navigation window:** Bring up the Navigation window under *Windows* > *Navigation.* The Navigation view contains three tabs:

a. **Object**: Set individual objects to navigation static, to generate OffMeshLinks and set their area type.

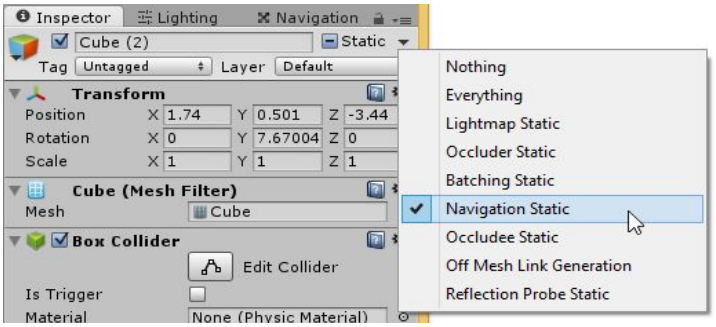
b. **Bake**: Here you define the dimensions of the agent. This helps Unity determine which areas of the level are accessible for this agent. In the advanced tab we can control voxel size and the minimum area required for a NavMesh to be generated.

c. **Areas**: Define custom area types and assign a cost to them. This will influence the path finding later on.

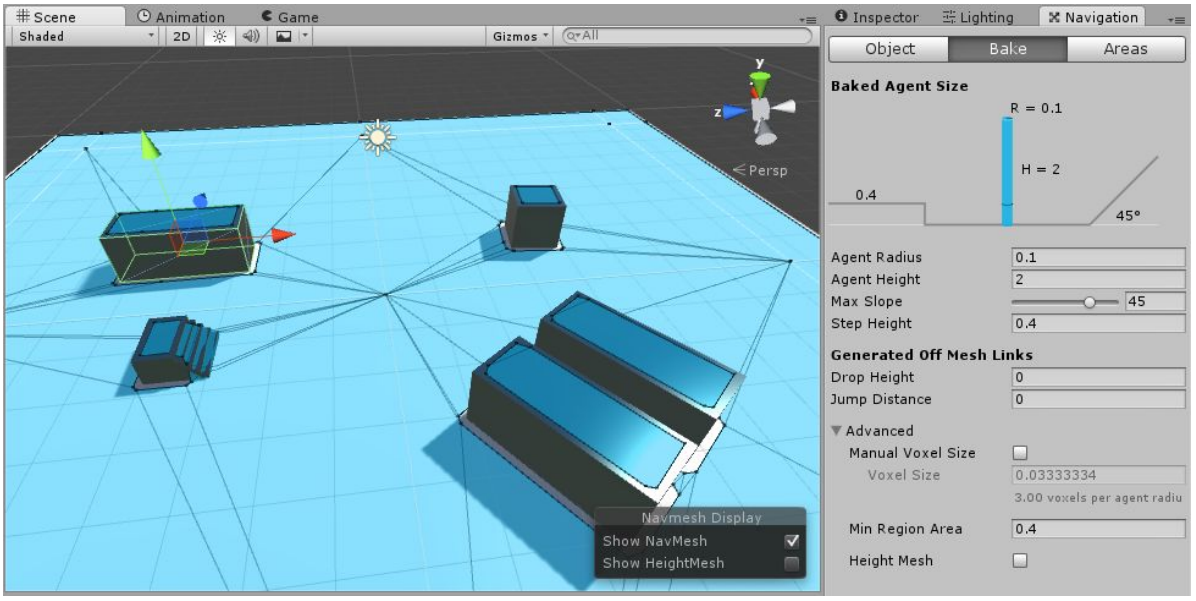


4. **Mark scene objects as Navigation Static:** You can either do this by checking the checkbox in the top right corner of the *Inspector*. Or in the *Navigation* view under the

Object tab.



5. **Adjust Baked Agent Size and bake:** Adjust the agent dimensions to fit the requirements of your level. Press the Bake button at the bottom of the Navigation window.

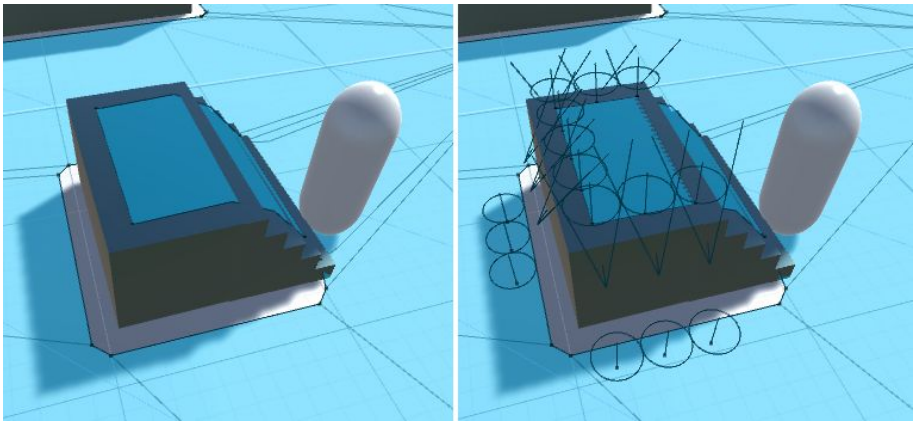


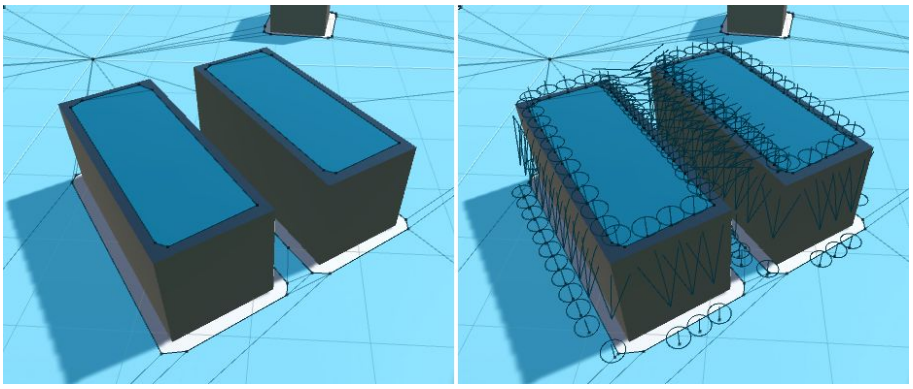
3.1.1 Unity Off-mesh Links

Off-mesh links are what you use If you want your agents to be able to jump gaps or drop down from elevated areas. They do exactly what their name suggests, they provide a link off the *NavMesh*, connecting a gap or two completely separate patches.

Here is what you have to do to generate *Off-mesh links*:

1. **Select the object that you want to generate off-mesh links from.**
2. **In the Navigation view check the Generate OffMeshLinks checkbox.**
3. **Adjust the Generate Off Mesh Links settings in the Bake tab.**
4. **Bake.**



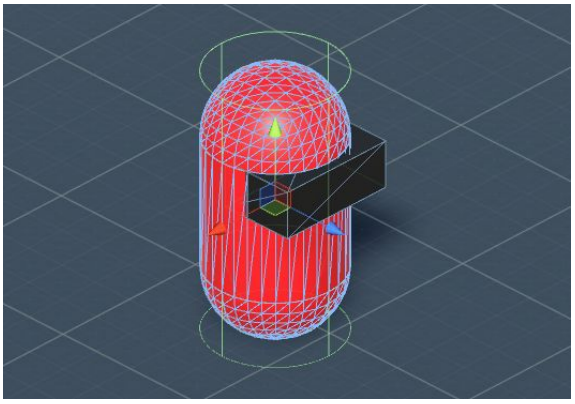


3.2 Unity NavMesh Game Object Components

3.2.1 Unity NavMesh Agent

The *NavMesh Agent* component is now the logic that applies the navigation and pathfinding functionality to a game object:

1. **Add a Capsule 3D object to act as an agent.**
   a. Create a red material and assign it to the agent.
   b. Add a cube as a child to the agent.
   c. Scale the cube down, so that it looks like the glasses of the agent.
   d. Give the glasses a black material and let them look to the z-direction:



2. **Add the NavMesh Agent component to the object.**

3. **Add script to agent object:** For our agent to be able to move through our level we have to actually set his destination. Add the following script to your agent object.

```
public class AgentBehaviour : MonoBehaviour
{
    public Transform target;
    NavMeshAgent _agent;

    void Start ()
    {   _agent = GetComponent<NavMeshAgent>();
    }

    void Update()
    {   _agent.SetDestination(target.position);
    }
}
```

4. **Assign the target public property:** Create an empty game object to act as a target transform for your agent.

5. **Play**. The agent should follow Sintel pretty quickly.

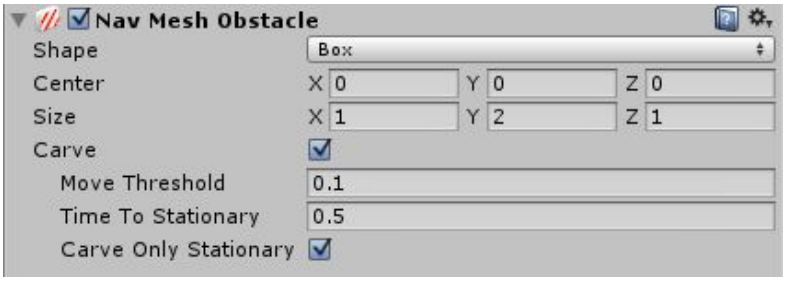
   If needed you can extend the agent script by utilizing the functions of the *NavMesh* class. http://docs.unity3d.com/ScriptReference/NavMesh.html

3.2.2 Unity NavMesh Obstacles

The *NavMesh Obstacle* is a component for dynamic objects that can block an agent's path. A barrel controlled by the physics system is a good example for such an obstacle.

The obstacle can either be defined by a box or a capsule based on the selection in the drop down.

It is recommended to turn on the *Carve* option. This will curve a hole into the *NavMesh* if the obstacle is stationary. Alternatively the *Carve Only Stationary* option can be turned off after which the carved hole is recalculated after the *Move Threshold* is exceeded.



3.3 Creating a patrolling AI that detects the player

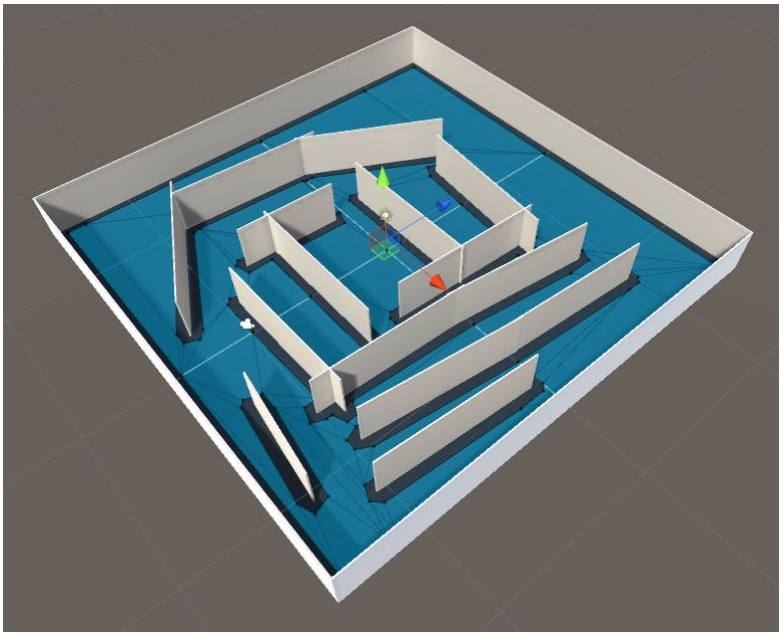
Our agent has now quite some super natural capabilities. He can find our target optimally even if he can't see it. We have to add now some artificial intelligence (AI) to make the finding more realistic.

1. We will add first a path following functionality to let the agent patrol along a given path.
2. Then we will simulate a visual detection from the agent to the target.
3. Optionally you could also add the hearing capability to improve the agent's recognition.

3.3.1 Patrol Path

1. **Duplicate the scene** in your *03 Navigation* project and call it *Scene-AI.unity*.

2. **Create an environment:** We need an environment for our agent to move around in. You can use simple cubes and planes to build a sufficient environment.
   Bake the *NavMesh* for your level after you're done.



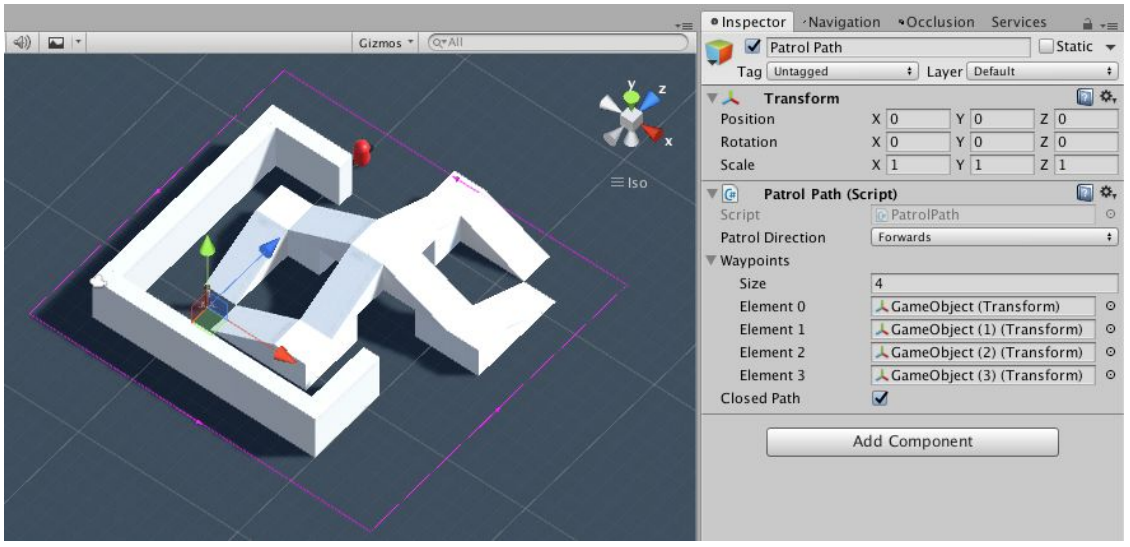
3. **Copy the *PatrolPath.cs* script** from Dropbox folder into your Scripts folder.
   The *PatrolPath* class implements a path with multiple waypoints that can be closed or not. It can be used by an agent to follow this path by setting the destination of the NavMesh agent to the next waypoint. If the *patrolDirection* is forwards an agent will receive with the method *SetNextWaypointIndex* the next higher waypoint index. If the *patrolDirection* is backwards it will get the previous waypoint index in the list.

4. **Create a path with multiple empty game objects**:

   a. Create an empty game object, call it *Patrol Path* and add the *PatrolPath* script to it.

   b. Add multiple empty game objects as children for the waypoints of the path.

c. Drag the way points to the waypoints list in *PatrolPath*.



d. **Create a new script for the agent and name it *AgentAIBehaviour***:
It lets the agent follow the path by setting the destination of the NavMesh agent to the next waypoint if he is closer to the current destination point than a given threshold (*reachedDistance*):

```
public class AgentAIBehaviour : MonoBehaviour
{
    public PatrolPath path;
    public float patrolSpeed = 3.5f;
    public float reachedDistance = 1.2f;
    private NavMeshAgent _agent;

    void Awake() {_agent = GetComponent<NavMeshAgent>();}

    void Update() {Patrol();}

    void Patrol()
    {
        _agent.speed = patrolSpeed;
        Vector3 dest = path.currentWaypoint.transform.position;

        _agent.SetDestination(dest);

        // Set the next destination point if we are closer than a threshold
        if (Vector3.Distance(dest, transform.position) < reachedDistance)
            path.SetNextWaypointIndex();
    }
}
```

e. **Set the *PatrolPath* reference** in the *AgentAIBehaviour* component to the path you just created.

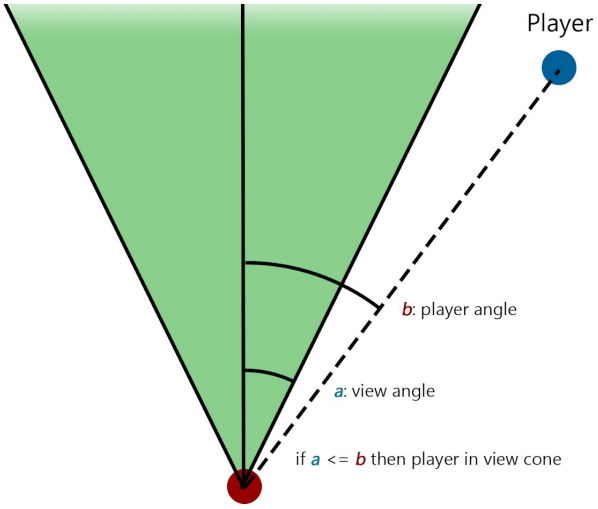
f. **Play!** Due to the given waypoints as new destinations the agent will smoothly follow the given path and not stupidly follow its lines.

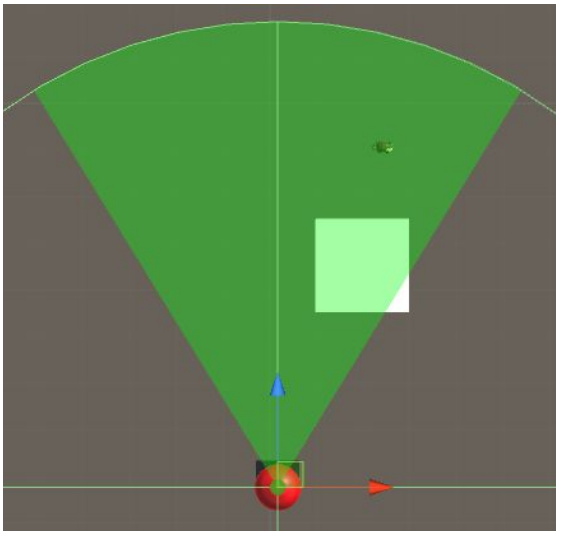
3.3.2 Adding Sight to the Agent

1. **Create an empty game object** on the agent without any transform and call it *Sight*.

2. **Add the script *AgentSight*** from the Dropbox to the *Sight* object.
The *AgentSight* class simulates the agent's sight by using a sphere collider to determine

if the player is closer than the colliders radius. If so, it checks whether the player is inside the view cone defined by the look direction and a view angle.
If so, it checks whether the player is actually visible by casting a ray to the player.
If the agent survives all tests he has direct sight to the player and we can call the event handler *playerSpottet* with the player's position.





3.3.3 Adding Chase and Searching States

In our *AgentAIBehaviour* script we can add the event handler *playerSpotted* and add two additional states beside *Patrolling*:

- **Chasing**: From *patrolling* we switch into the *chasing* state as soon as the player got spotted. We set the destination to the *lastPositionOfInterest* and increase the speed. The *NavMesh* AI then guides the agent towards the player. The agents *alertLevel* is set to the max. value 1.
- **Searching**: If the player gets out of sight during *chasing* we switch to the *searching* state. For a certain time the agent continues its path to the *lastPositionOfInterest*, its *alertLevel* and speed is decreasing. If his *alertLevel* becomes zero we switch back to the *patrolling* state.

1. Add the following variables to the *AgentAIBehaviour* script.

```
public enum State
{   Patrolling,
    Chasing,
    Searching
}
public AgentSight   sight;
public float        chaseSpeed = 6.0f;
public float        alertLevelDecreaseTime = 10.0f;
private State       _state = State.Patrolling;
private float       _alertLevel = 0.0f;
private Transform   _player;
private Vector3     _lastPositionOfInterest;
```

2. In the *Awake* method we also retrieve a reference to the player's transform:

```
_player = GameObject.FindGameObjectWithTag("Player").transform;
```

3. Connect the *playerSpotted* callback in the Enemy script:

```
void OnEnable()  {sight.playerSpotted += PlayerSpotted;}
void OnDisable() {sight.playerSpotted -= PlayerSpotted;}
```

```
void PlayerSpotted(Vector3 position)
{   _state = State.Chasing;
    _lastPositionOfInterest = position;
}
```

4. In the *Update* method we switch to the new *states*:

```
void Update()
{   switch (_state)
    {   case State.Patrolling: Patrol(); break;
        case State.Chasing:    Chase();  break;
        case State.Searching:  Search(); break;
    }
}
```

5. Add the Chase handling method:

```
void Chase()
{
    // if player is not in sight and we arrived at the last position he was seen at
    // then switch to searching the area
    if (!sight.playerInSight && _agent.remainingDistance <= _agent.stoppingDistance)
    {   _state = State.Searching;
        return;
    }

    // always look at the current player position while chasing
    sight.LookAtPosition(_player.position);

    // update speed
    _agent.speed = chaseSpeed;

    // update agent destination
    _agent.SetDestination(_lastPositionOfInterest);

    // keep alert level on maximum
    _alertLevel = 1.0f;
}
```

6. Add the Search handling method:

```
void Search()
{
    // set speed to a lerp between chase and patrol speed based on alert level
    _agent.speed = patrolSpeed + _alertLevel * (chaseSpeed - patrolSpeed);

    // decrease alert level over time
    if (alertLevelDecreaseTime > 0.0f)
         _alertLevel -= Time.deltaTime / alertLevelDecreaseTime;
    else _alertLevel = 0.0f;

    if (_alertLevel <= 0.0f)
    {   _alertLevel = 0.0f;
        _state = State.Patrolling;
    }

    // keep looking at the current target
    sight.LookAtPosition(_lastPositionOfInterest);
    _agent.SetDestination(_lastPositionOfInterest);
}
```

7. Add the following line to the *Patrol* state method:

```
// Reset the look direction to the z-axis
sight.LookForward();
```

8. And finally add a few lines to show the agents view cone in the editor:
For the *Handles* reference you need to insert *using UnityEditor;* at the top of the script.

```
void OnDrawGizmos()
{
    if (sight == null) return;

    // project the world space head direction onto the xz-plane (normale = up-axis)
    Vector3 headDirProj = Vector3.ProjectOnPlane(sight.headLookWS, Vector3.up);

    // add a half view angle rotation to center the view cone around the up-axis
    Quaternion rot = Quaternion.AngleAxis(-0.5f * sight.viewAngle, Vector3.up);

    // lerp the color based on alert level
    Handles.color = Color.Lerp(new Color(0.0f, 1.0f, 0.0f, 0.2f),
                               new Color(1.0f, 0.0f, 0.0f, 0.2f), _alertLevel);

    Handles.DrawSolidArc(transform.position,
                         Vector3.up,
                         rot * headDirProj,
                         sight.viewAngle,
                         sight.viewRadius);
}
```

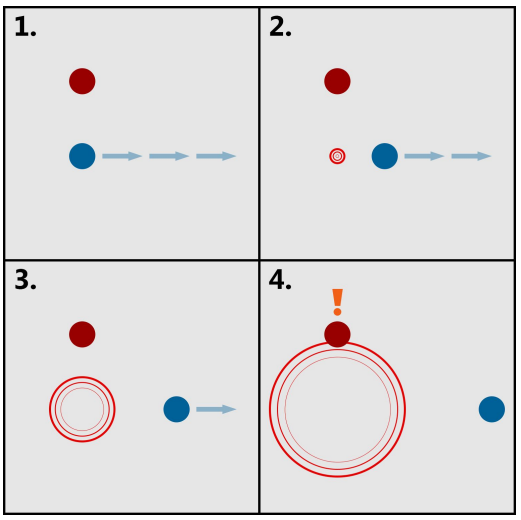
9. Tag the Sintel character with the *Player* tag.

10. Play!

Checkpoint 8: Patrolling AI

3.3.4 Adding Hearing to the Agent

There are many different ways of achieving similar results. The way we will be doing it is mostly for visual appeal and simplicity. You could go further and simulate sound waves reflecting from walls until they reach the enemy. Or do it even simpler by defining a set radius on an enemy where he's able to hear the player move.
In the image below you're able to see how our implementation will work. The player emits a sound wave object in a regular interval. These sound waves will expand over time and if they reach an enemy the enemy will be alerted about the sound. We set the radius of the soundwave dynamically based on how fast the player is moving. The faster the player is moving the 'louder' the sound waves he produces.



a. Create a new C# script and call it EnemyHearing.

```
[RequireComponent(typeof(SphereCollider))]
public class EnemyHearing : MonoBehaviour
{
    public delegate void SoundHeardDelegate(Vector3 position, float falloff);
    public SoundHeardDelegate soundHeard;

    void OnTriggerEnter(Collider other)
    {
        if (other.tag != "Soundwave")
            return;
        Soundwave soundwave = other.GetComponent<Soundwave>();
        if(soundwave != null && soundHeard != null)
            soundHeard(other.transform.position, soundwave.normalizedVolume);
    }
}
```

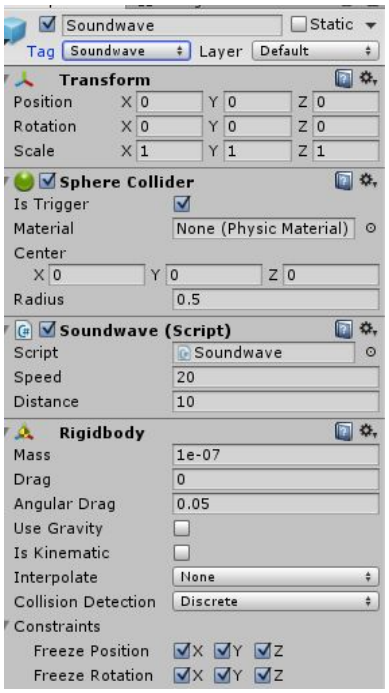
b. Just as you did with EnemySight, add a child GameObject to your enemy and call it Hearing. Attach EnemyHearing to the object.

c. Create a new C# script and call it Soundwave

```
[RequireComponent(typeof(SphereCollider))]
public class Soundwave : MonoBehaviour
{
    public float speed;
    public float distance;
    private float _radius = 0.0f;
    private SphereCollider _trigger;

    // normalized sound volume in range [0, 1]
    public float normalizedVolume {get { return 1.0f - _radius / distance;}}

    void Start()
    {   _trigger = GetComponent<SphereCollider>();
        _trigger.isTrigger = true;
        _trigger.radius = 0.0f;
    }

    // Update is called once per frame
    void Update ()
    {   _radius += Time.deltaTime * speed;
        _trigger.radius = _radius;
        if (_radius > distance)
```

```
            Destroy(gameObject);
        }
    }

    void OnDrawGizmos()
    {
        Gizmos.color = new Color(1.0f, 1.0f, 0.0f, 0.1f);
        Gizmos.DrawSphere(transform.position, _radius);
    }
```

d. Create an empty GameObject and call it Soundwave, attach the Soundwave script to it. Add a Rigidbody component too and copy the settings you see in the screenshot below.
The rigidbody is needed because the Trigger functions of a MonoBehaviour (OnTriggerEnter etc.) only get called if one of the interacting colliders has a rigidbody attached.



e. Drag the Soundwave GameObject into a Folder in your Project view to create a Prefab. You can delete the Soundwave object in the scene again.

f. Update the PlayerControl script to emit *Soundwaves* while moving:

```
public Soundwave soundwave;
private float _footstepInterval = 0.5f;
private float _footstepTimer = 0.0f;
```

g. In the Update method of the PlayerControl script add the following lines:

```
// emit a soundwave every _footstepInterval
_footstepTimer += Time.deltaTime;
if (_footstepTimer > _footstepInterval && _velocity.sqrMagnitude > 0.0f)
{
     _footstepTimer = 0.0f;
     EmitSoundwave();
}
```

h. Add the EmitSoundwave method:

```
void EmitSoundwave()
{
    Soundwave sound = Instantiate(soundwave,
```

```
                                    transform.position + Vector3.up * 0.1f,
                                    Quaternion.identity) as Soundwave;
        sound.distance = 10 * (_velocity.magnitude / maxVelocityRangeEnd);
    }
```

i. Drag the Soundwave Prefab into the Soundwave reference property on the PlayerController script.

j. Update the Enemy script with the following code

```
// reference to the enemy hearing gameobject
public EnemyHearing hearing;

// ratio of sound volume heard to alert level gained
float soundAlertLevelIncrease = 1.0f;
```

k. Attach the SoundHeard callback

```
// connect the sight and hearing callbacks on enable
void OnEnable()
{
    sight.playerSpotted += PlayerSpotted;
    hearing.soundHeard += SoundHeard;
}
// disconnect the sight and hearing callbacks on disable
void OnDisable()
{
    sight.playerSpotted -= PlayerSpotted;
    hearing.soundHeard -= SoundHeard;
}
// called whenever this enemy hears a sound
void SoundHeard(Vector3 position, float volume)
{
    // update the last position of interest
    _lastPositionOfInterest = position;

    // increase alert level relative to the volume of the sound
    _alertLevel += soundAlertLevelIncrease * volume;
    _alertLevel = Mathf.Clamp(_alertLevel, 0.0f, 1.0f);

    // search the area the sound came from if we're not chasing anyone
    // and our alert level is high enough
    if (_alertLevel > 0.5f && _state != State.Chasing)
        _state = State.Searching;
}
```

l. Play.

Temporary notes 'n stuff:

Resources:

https://unity3d.com/learn/tutorials/topics/navigation

https://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/state-machine-interface

https://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/navmeshes

http://unity3d.com/learn/tutorials/projects/stealth/enemy-ai

4 Advanced Visual Effects

In the predecessor module *Game Development* we discussed the lighting topic in Unity only very briefly. In chapter 2.4 of the *Game Development* script we looked at the new *Standard Shader* that was introduced with Unity 5. We also used the new *Global Illumination* features of Unity 5 just as is without any background information. In this chapter we want to take a look at how these features can be used to create very appealing lighting effects.

4.1 Lighting Effects

You can find a good overview on Unity's lighting capabilities in the Lighting Overview Video.

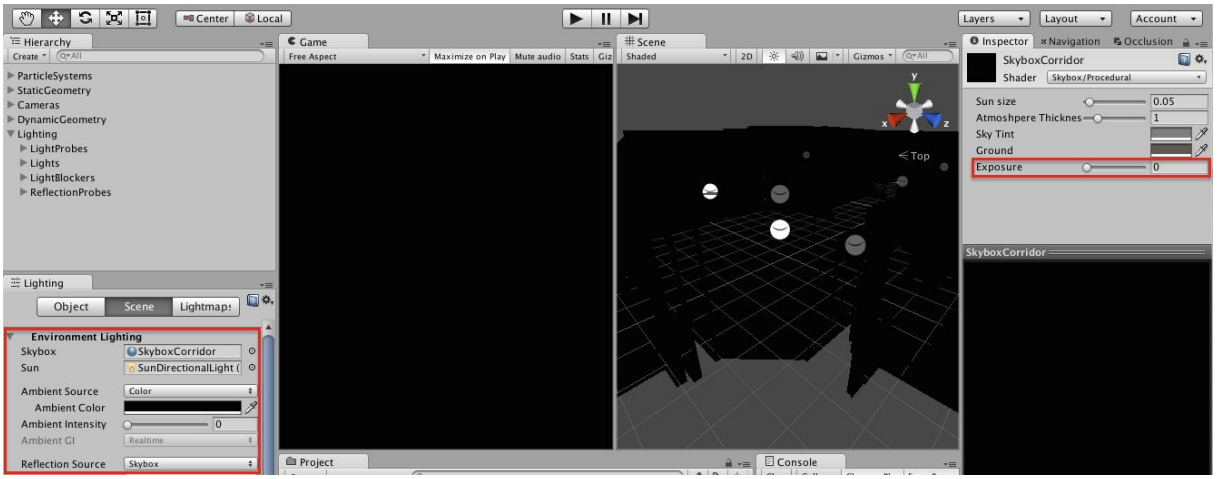
This tutorial is based on Unity's Corridor Lighting Example and explains classic lighting as well as some advanced lighting effects. In contrast to the project from the Unity asset store we will use the project with all lighting features turned off. We will then turn them back on again one after the other.

1. Copy the *4 Advanced Visual Effects* folder from the Dropbox folder to your file system.
2. Open the Unity be double-clicking the file *../Assets/Scenes/CorridorDemo.unity*
3. The scene as well as the Game window will be completely black because all light sources are turned off.
4. Reduce the Gizmo's size with the slider.

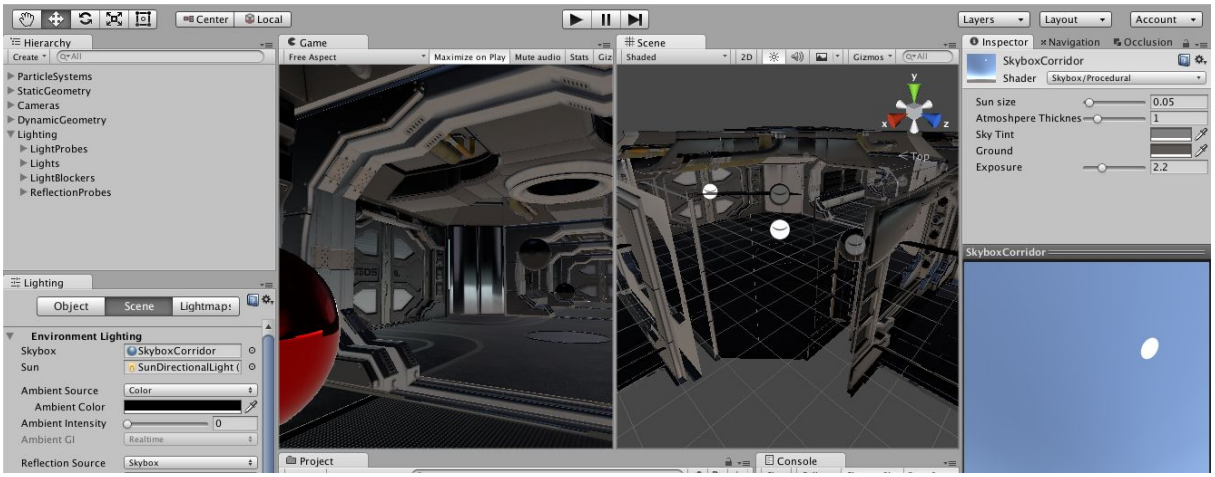
4.1.1 Environment Light

Since Unity 5 we can use the skybox as an environment light source.

1. **Turn the skybox light source on & off again**: If we open the *Lighting* window with the menu *Window > Lighting* we see that the scene uses the standard skybox for the environment lighting.



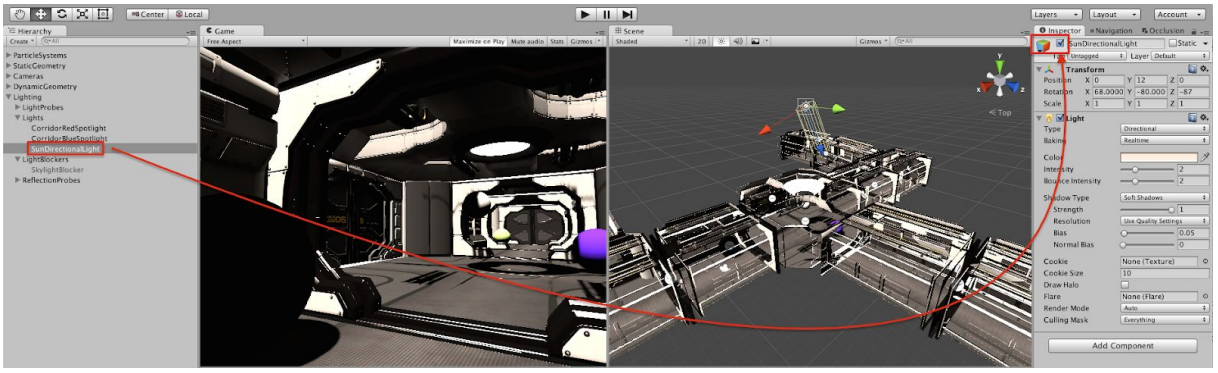
If we set the Exposure of the skybox to a higher value than zero we see the environment light reflected.



But because the model is a space station there is no illumination from the sky and we can turn the exposure back to zero.

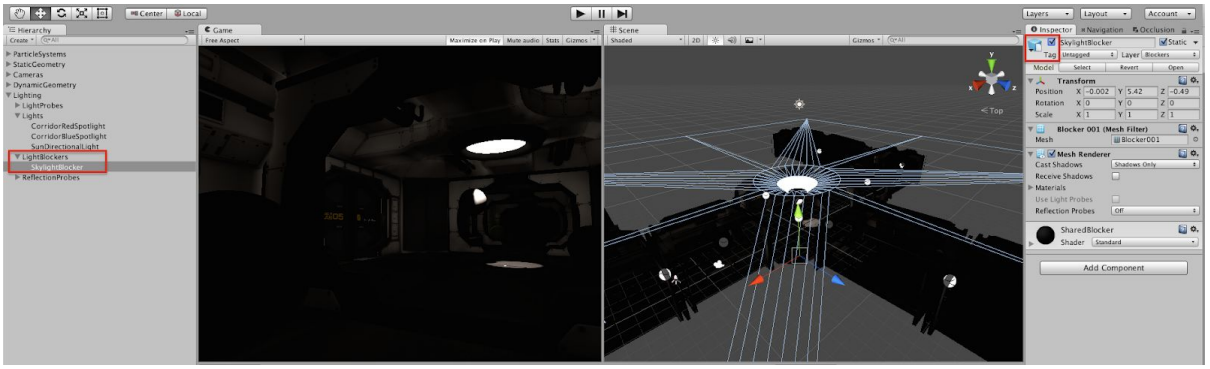
4.1.2 Direct Light Sources

1. **Turn on the directional light source for the sun light**: Enable the *Lighting/Lights/SunDirectionalLight*. For the first time we see the space station model with classic direct lighting from a directional light source.



If you want to know more about classic lighting in computer graphics you can read chapter 8 in the *CPVR-ComputerGraphics-Script.pdf* in the Dropbox folder.

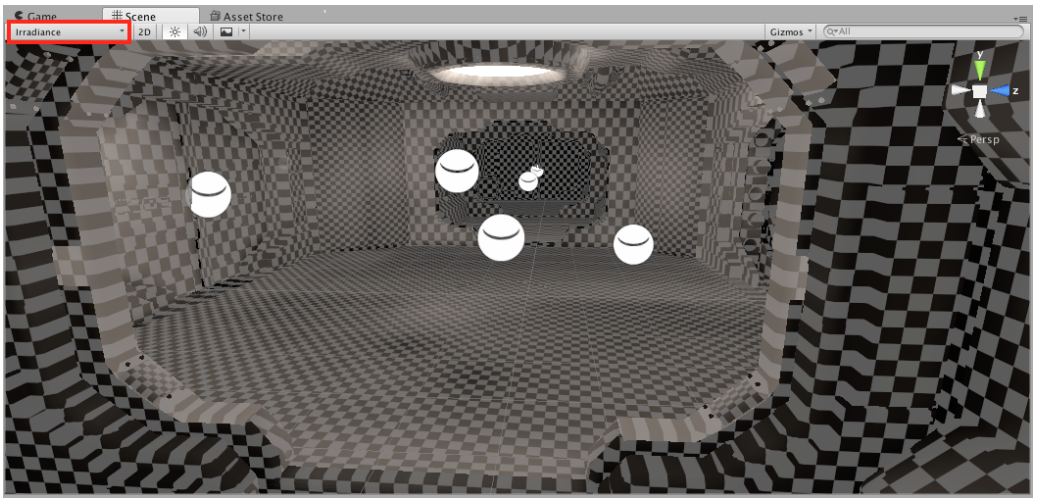
2. **Turn on the Skylight Blocker**: Because the shadowing process does not block the direct light if it hits a mesh from the backside the space station is illuminated almost everywhere. To avoid that enable a special *Lighting/LightBlockers/SkylightBlocker* mesh. In the *Mesh Renderer* it uses the *Cast Shadows* option *Shadows only*. After enabling it you will see that the station has only one round window where the sun light can enter:



You should be able to see a minimal effect of the global illumination (GI) process activated in the *Lighting* window. Some surrounding walls receive indirect lighting

reflected from the white sphere and from the ground. The GI calculation can last several minutes depending on the laptop's performance.

3. **Adjust the Realtime Resolution** in the Precomputed Realtime GI section. The GI calculation depends strongly on the resolution of the light maps that are generated in the background. You can view the resolution of the irradiance maps:



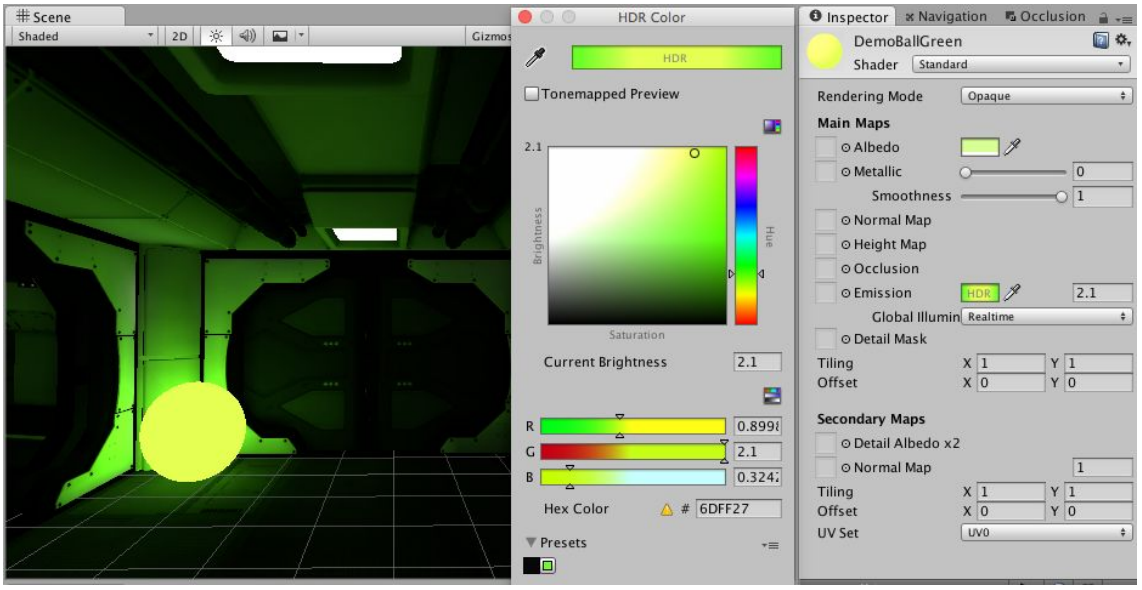
4. **Turn on the red and blue spot lights**: Even if you turn on these direct lights the illumination remains very weak. There must be some other light sources!

4.1.3 Material Light Sources

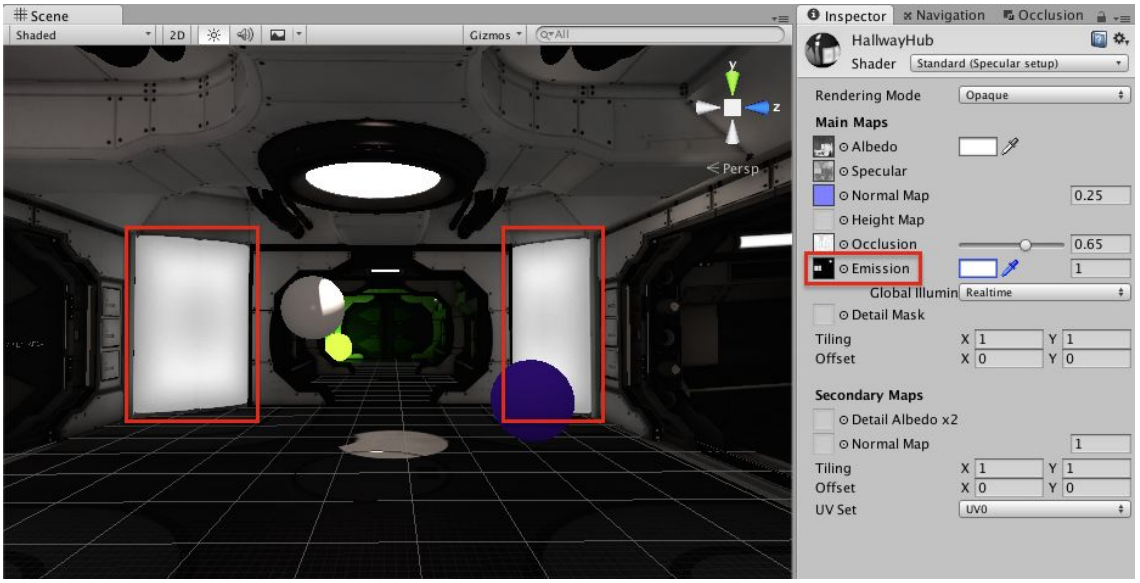
Also since Unity 5 we can specify any material as an emissive light source.

1. In the *Project* window set the *Emission* value of the following materials:
   - *Material/Hallways/HallwayDownlighter* to 1.0
   - *Material/Hallways/HallwayHub* to 1.5
   - *Material/Hallways/HallwayWall* to 1.0
   - *Material/DemoBalls/DemoBallGreen* to 2.1

2. Set the ***Emission* Color** of the *DemoBallGreen* material to a light green:



3. **Emission Texture**: As you can see on the *HallwayHub* material you can also define a texture image to restrict the light emission to certain mesh areas:



4.1.4 Reflection Probes

A *Reflection Probe* is an invisible camera that captures a spherical view of its surroundings in all directions. The captured image is then stored as a cubemap that can be used by objects with reflective materials. Several reflection probes can be used in a given scene and objects can be set to use the cubemap produced by the nearest probe.

1. **Reflection Probes strongly depend on Project Settings**:

   a. **Quality Settings**: Under *Edit* > *Project Settings* > *Quality* you can turn on or off the *Realtime Reflection Probes*.

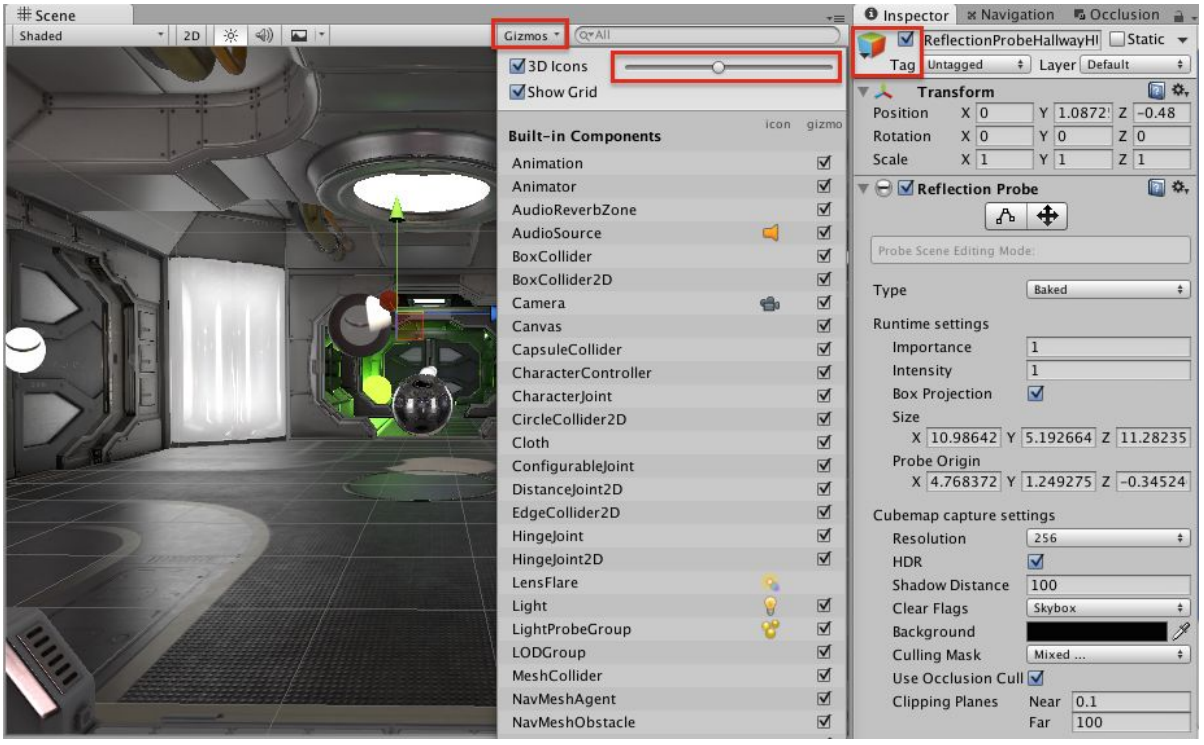
   b. **Rendering Path**:

      i. **Deferred**: This can be set under the *Player Settings* (Edit > Project Settings > Player) in the subsection *Other Settings*. From the Unity manual we learn that there is no limit on the number of lights that can affect a GameObject when we use the deferred rendering path. All lights are evaluated per-pixel, which means that they all interact correctly with normal maps, etc. Additionally, all lights can have cookies and shadows. On the downside, deferred shading has no real support for anti-aliasing and can't handle semi-transparent GameObjects. There is also no support for the *Mesh Renderers Receive Shadows* flag.

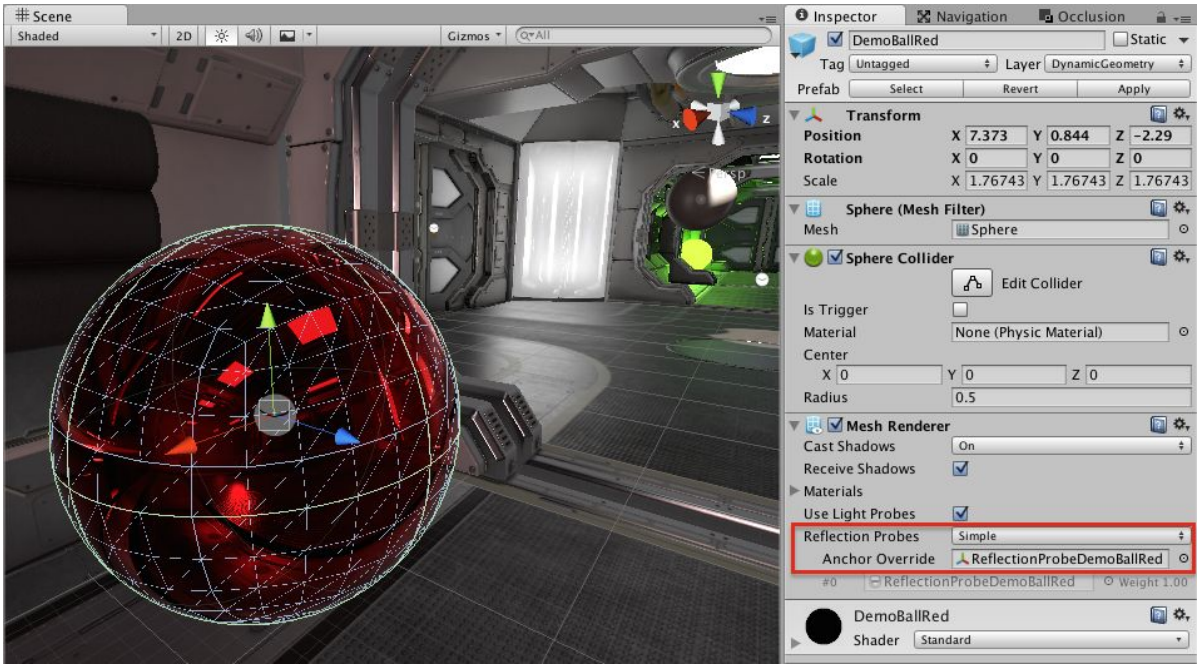
      ii. **Forward**: This is the old school render path where the performance is linear to NO. of lights in the scene. Read more in the Unity manual.

2. **Turn on all reflection probes** in the scene hierarchy under
   - Lighting/ReflectionProbes
   - DynamicGeometry/DynamicBallWhite
   - DynamicGeometry/DynamicBallPurble
   - DynamicGeometry/DynamicBallRed

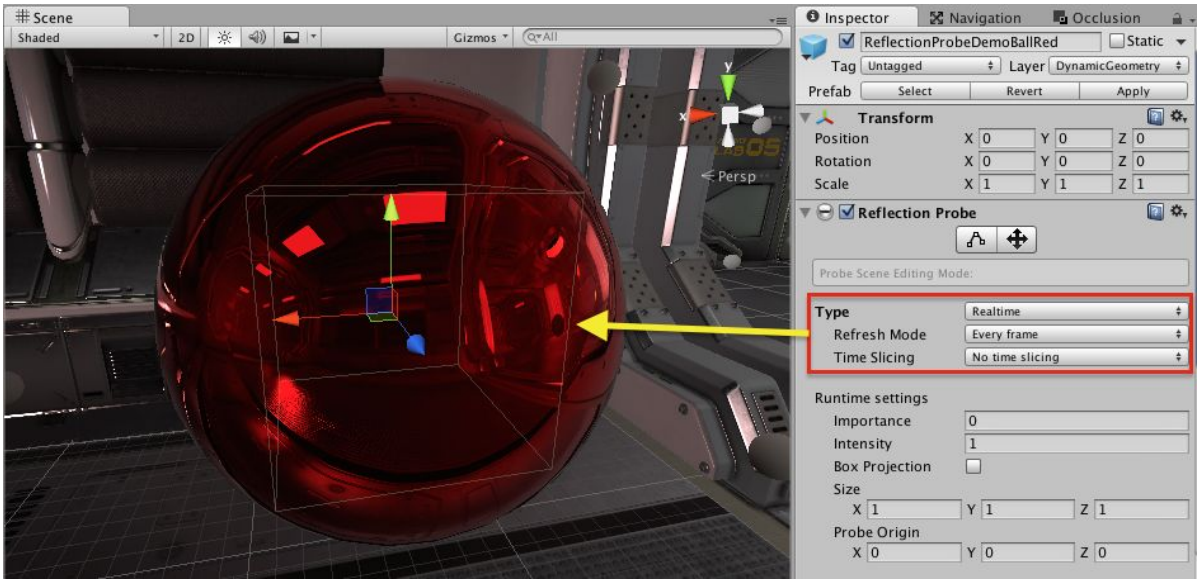
3. **Visualize with Gizmos**: Reflection probes are invisible in the Game but can be visualized in the Scene window with *Gizmos*:



4. **Mirror Balls**: With the *ReflectionProbeDemoBallRed* inside the *DemoBallRed* object you can create a perfectly mirrored sphere. The reflection probe has to be assigned to the *Mesh Renderer* component of the game object:



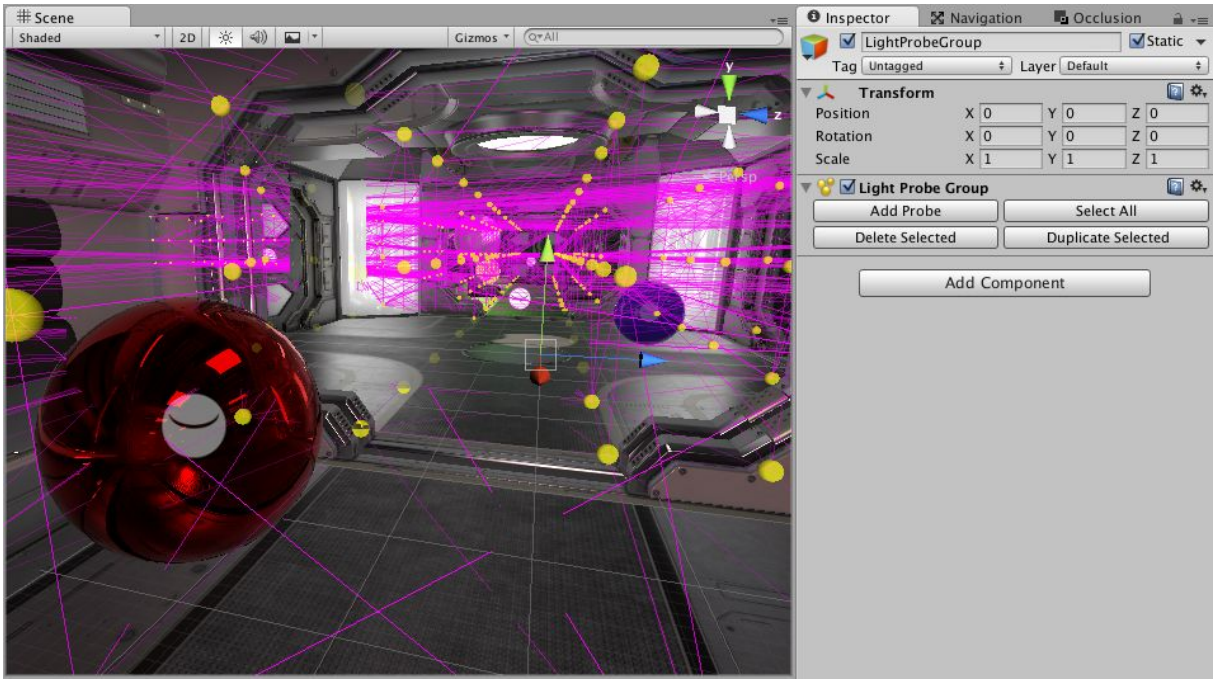
5. **Dynamic Reflections**: If you run the game you may notice that the bouncing white ball is not reflected in the red ball. You can achieve this by setting the red balls reflection probe type to *Real-time*. If you choose *Every Frame* as the *Refresh Mode* the reflection cubemaps will be generated every frame and will therefore drastically reduce the frame rate.



4.1.5 Light Probes

As we can see indirect lighting through reflections has a large effect on the overall illumination of a scene. This indirect illumination is calculated in the global illumination (GI) preprocess. This is a demanding process and can take a few seconds before becoming visible in the scene view. It applies only to static objects. For dynamic object such as the three spheres (*DynamicBallWhite*, *DynamicBallPurble*, *DynamicBallRed*) the GI process can be simulated with so called *Light Probes*. The idea is that the lighting is sampled at strategic points in the scene, denoted by the positions of the probes. The lighting at any position can then be approximated by interpolating between the samples taken by the nearest probes.

1. **Turn on the Light Probes**: As you can see the density of the light probes is quite high.



4.1.6 Particle Systems

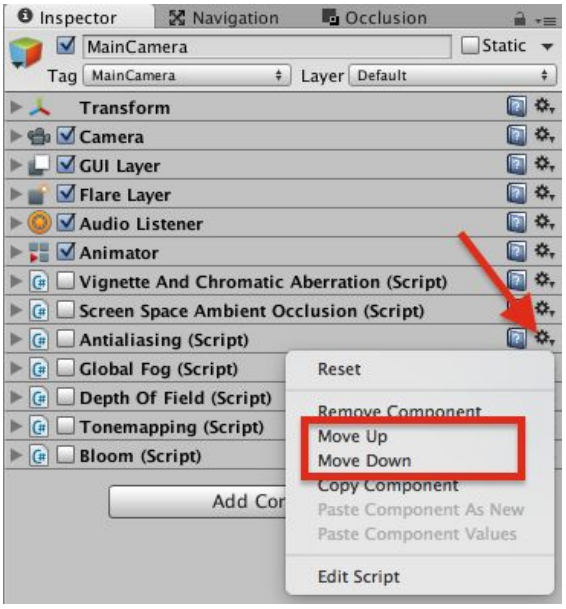
To complete Unity's *Corridor Lighting Example* project we turn the particle systems that add some dynamic haze into the scene back on.

4.2 Image Effects

This chapter is also based on Unity's Corridor Lighting Example and explains advanced effects that can be applied as post processing filters on the camera object. The image effects within the Corridor Lighting Example are also explained in Unity's Image Effects Overview video. We therefore omit a step by step explanation. Details on different image effects and their parameters can be found in the Unity documentation.

- **Adding Image Effects**: Image Effects have to be imported first with *Assets* > *Import Package* > *Effects*. Afterwards they can be added to a camera object as a component with the menu *Component* > *Image Effects* > … .

- **Execution Order**: Image effects are C# scripts that are executed after the frame has been rendered in the order they appear in the inspector window. You can change the order in the settings menu of the component:



- **Performance Impact**: All image effects are executed on every frame and can therefore have a big impact on the performance. You can check the impact of an effect best by supervising the framerate in the Stats panel.

5 Performance Optimization

In this chapter we want to focus on the optimization of the game performance. It is difficult to understand the key factors that influence the performance without a deep bottom up understanding of the entire real-time rendering pipeline. We therefore will begin with a brief introduction on the *Real-Time Rendering Pipeline*.

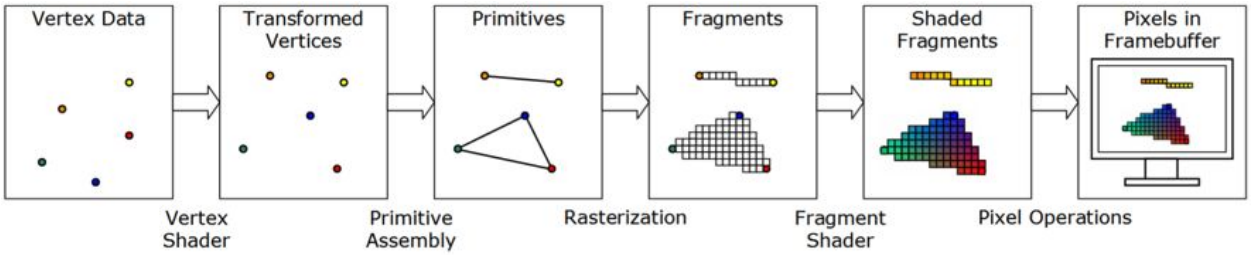
- Real-Time Rendering Pipeline
- Performance Measures
- Profiling Tools
- Optimizations
    - Draw less
        - Occlusion Culling
        - Level of Detail
    - Draw faster
        - Batching
        - Script Optimization

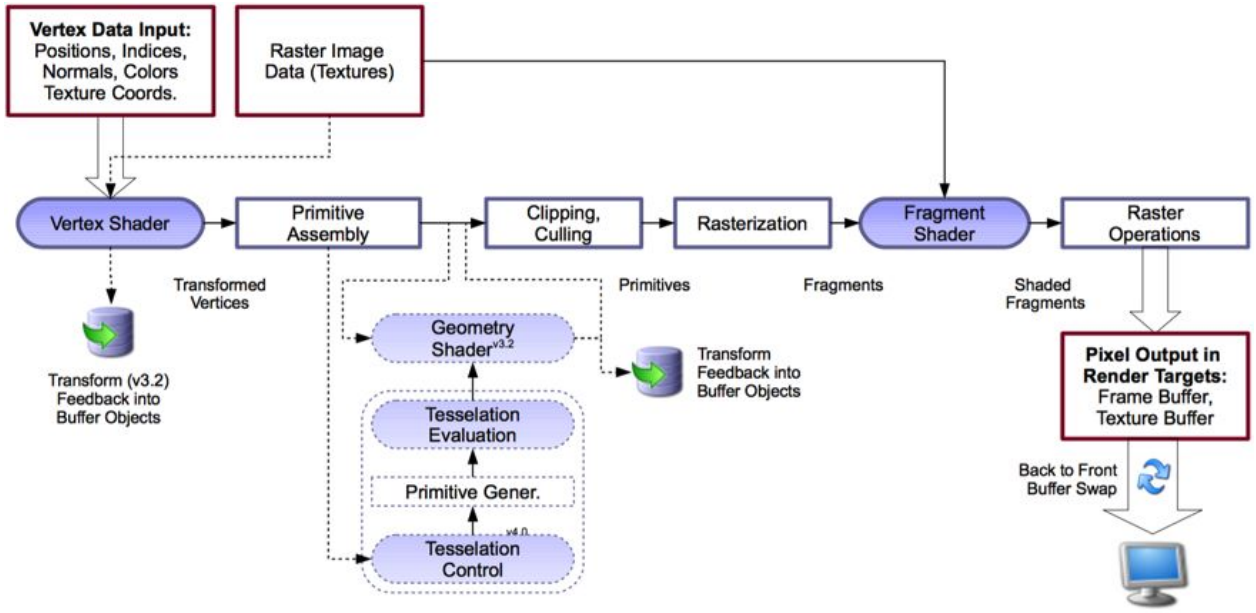
5.1 Real-Time Rendering Pipeline

The real-time rendering pipeline is the process with the 3D data on the input side and the image of frame buffer on the output side. This process chain is highly optimized so that the framebuffer can be filled around 60 times per second. To achieve this the CPU has a very powerful helper, the GPU (Graphics Processing Unit). The GPU can be on the same processor or on a dedicated graphics card.

Since around 10 years we can program specific processing steps on the GPU to achieve certain visual effect. These so called *shader programs* are mostly short programs on specific processing location within the pipeline. Their input and output is always the same, so that they can be executed in parallel. The most important shader programs are:

- **Vertex Shader**: The vertex shader is responsible to transform the vertex from their original 3D world position into their final position on the 2D screen. The vertex shader is executed for every vertex.
- **Fragment Shader**: The fragment shader is responsible for the final color of each pixel in the frame buffer. The fragment shader is executed for every pixel.





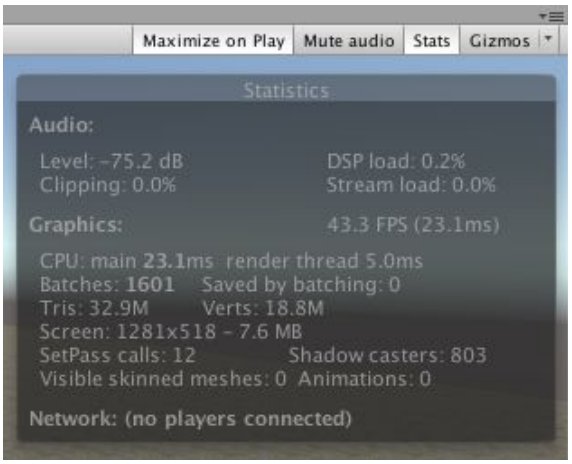
Schematic OpenGL core profile pipeline

Because the real-time rendering process is a pipelined chain of many processing steps there will always be one step that acts as the bottleneck. If this bottleneck is on the CPU side we call the process *CPU-bound*, if it is on the GPU we call it *GPU-bound*. There are many steps on both sides that can become a bottleneck. We will cover most of them in the following subchapters:

- **CPU-bound**:
  - Complex or inefficient game logic
  - Complex physics simulation
  - Long resource loading time
  - Too many animated characters (mesh skinning)
  - Too many or too large blend shapes
  - Too many or large resources (models & textures)
  - Too many draw calls
  - Slow network communication
- **GPU-bound**:
  - Too many vertices:
    - Large models
    - Dense particle systems
  - Too many pixels in large screens
  - Too complex calculations in the shader programs:
    - Complex vertex displacement
    - Complex lighting
    - Complex image effects

5.2 Performance Measures

You can show various performance measures in the *Statistics* overlay window by pressing the *Stats* button in the *Game* window:



During game play it is constantly refreshed and shows the following measures:

- **FPS**: The number of frames drawn per second in the *Game* window.
  In brackets is the time in milliseconds used for one frame (= 1/FPS)
- **CPU**: Time used for one frame on the CPU.
- **Render Thread**: Time used on the CPU side for the rendering thread that does the state changes and draw calls.
- **Batches**: A *Batch* is a synonym for a draw call of a 3D object with a certain material.
- **Saved by batching**: Number of draw calls saved by combining them into larger batches.
- **Tris and Verts**: The number of triangles and vertices drawn.
- **Screen**: The size of the screen and its memory usage.
- **SetPass**: The number of rendering passes.
- **Visible skinned meshes**: The number of skinned meshes rendered.
- **Shadow casters**: The number of object that can cast a shadow.
- **Animations**: The number of animations playing.

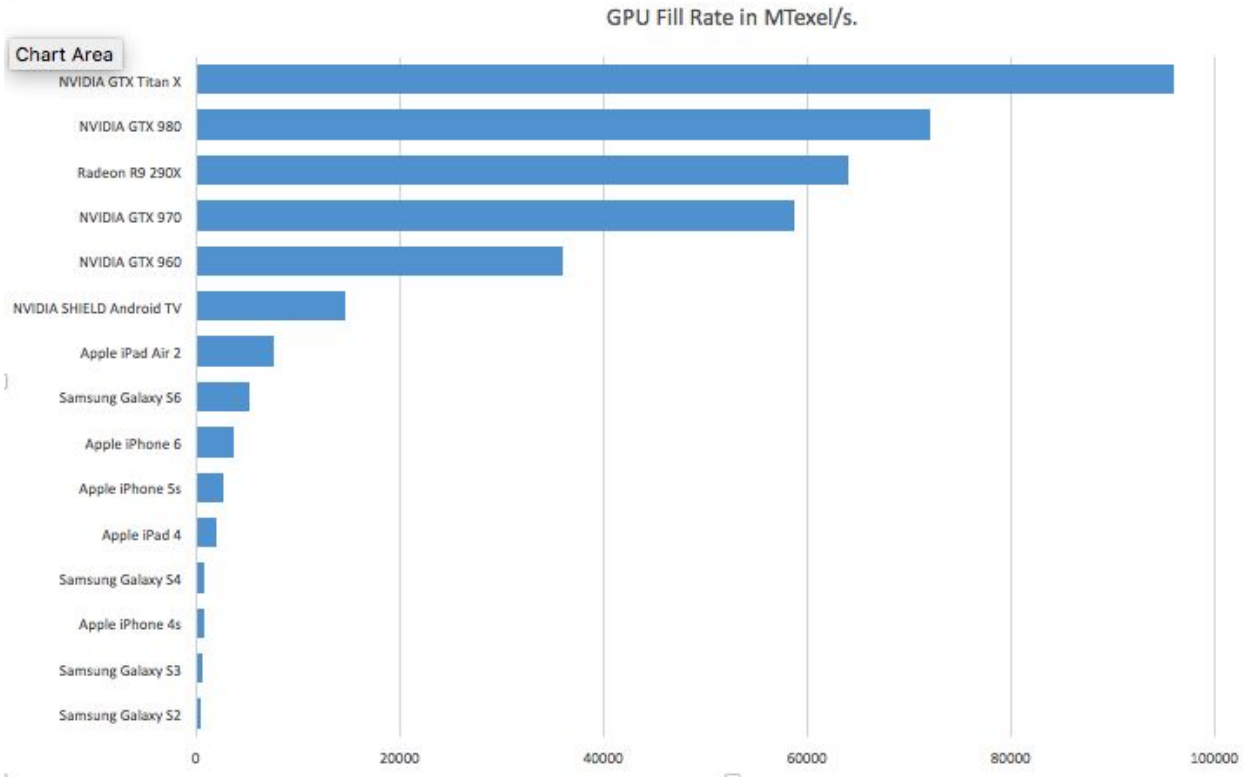
The raw performance of the GPU is often measured in the *fill rate* and the *triangle throughput*.

- **Fill Rate**: Pixels drawn per second
- **Triangle Throughput**: Triangles drawn per second.

These measures are rather theoretical but can be used for comparison:

- Theoretical Example: **iPhone 5s** (1136 x 640 pixels)
  - Fill Rate: 3304 MTex/s, Triangle Throughput: 68 MTri/s
  - Max. FPS from fill rate = (3300×10^6) / (1136 x 640 x 60) = 75.7 fps
  - Max. triangles per frame = (68×10^6) / 60 = 1.13 Mio.
- Theoretical Example: **iPhone 4** (960 x 640 pixels)
  - Fill Rate: 400 MTex/s, Triangle Throughput: 14 MTri/s
  - Max. FPS from fill rate = (400×10^6) / (960 x 640 x 60) = 10.9 fps
  - Max. triangles per frame = (14×10^6) / 60 = 0.33 Mio.

The following chart shows that the best mobile GPU's performance is roughly x10 below the one of the best dedicated desktop GPU:



5.3 Profiling Tools

Besides the *Statistics* overlay window there are other internal profiling and debugging tools:

5.3.1 Unity Profiler

The Unity Profiler gives very detailed information over time of a running game. You can setup different profilers for:

- CPU Usage
- GPU Usage
- Rendering
- Memory
- Physics 2D & 3D
- Network
- Audio

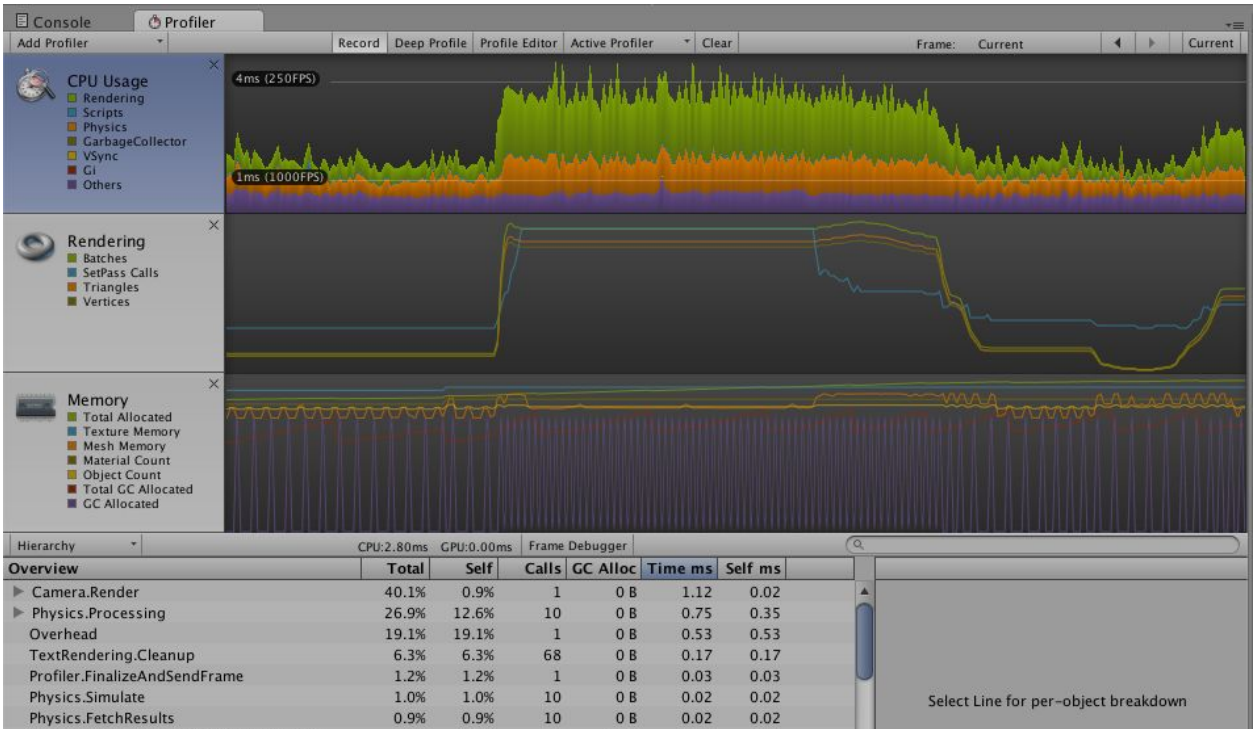
Multiple Measurements per Profiler: Each profiler shows multiple measurements that most can be hidden by clicking on the little square to the left of the measure name.

Sorted List of Measures: On the bottom left you can see the different measurement in a sorted list that can be expanded into deeper details.

Measure Details: On the bottom right you can see the details of the selected measurement.

Profile Mobile Devices: An important feature of the profiler is that it is able to be attached externally to the *WebPlayer* and *iOS* or *Android* device.

Unity Documentation: You can find more written information in the Unity documentation or in various video tutorials.



5.3.2 Unity Frame Debugger

The *Unity Frame Debugger* lets you step through the construction of a single frame. This is not directly related to improving the performance but it visualizes what happens step by step to build the frame. Mostly this will be drawing calls of the single objects but you will also see the shadow map creation into separate render targets.

See the Unity video tutorial on the Unity Frame Debugger for a good introduction.

5.3.3 Resource Checker

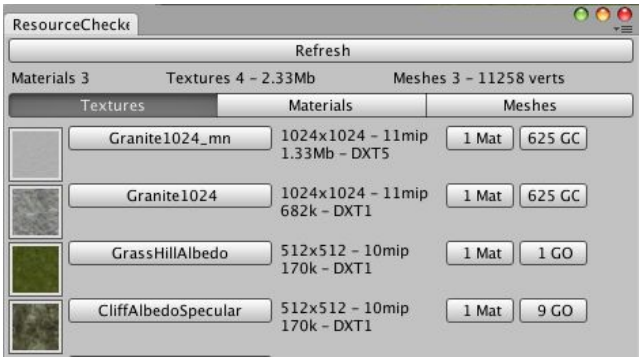
From the *Asset Store* you can download the free *Editor* extension *Resource Checker*.

Resource checker is designed to help bring visibility to resource use in your scenes (e.g. what assets are using up memory, which meshes are a bit too detailed, where are my materials and textures being used).

It should also be useful to check for redundant materials, textures you forget you're using, textures that can be compressed or reduced in size.

To use, just create an *Editor* folder within *Assets* in your project if you don't already have one, and drop *ResourceChecker.cs* in there.

In Unity you'll see a new option under *Window* > *Resource* Checker.



5.4 Optimizations

There are in general two strategies to improve the game's performance: Draw less or draw faster. In both strategies are many possibilities for bottlenecks:

- **Draw less**: In real-time CG with the pipeline explained at the beginning of the chapter the framerate is in general dependent on the
  - **No. of vertices** that you send through the vertex shader:
    - Draw only what you see:
      - *Frustum Culling*: Don't draw meshes outside of the camera's view frustum.
      - *Occlusion Culling*: Don't draw meshes that are occluded.
      - Draw really only what you will ever see.
    - Draw high resolution meshes only close to the camera: *Level of Detail*
    - Draw meshes in general with lower resolution:
      - Increase details with textures and bump mapping.
      - A *highpoly realistic style* is not a guarantee for a good game.
      - A *lowpoly artistic style* can lead to very fast and fun games.
  - **No. of pixels** processed in the fragment or pixel shader:
    - Reduce screen size
  - **No. of textures** and their size have an impact on the performance if the memory on the GPU gets used up:
    - Reduce texture size and bits per pixel
    - Use compressed texture formats

- **Draw faster**: This is a lot less obvious than drawing less and needs a good knowhow on low end CPU & GPU technology and programming:
  - **GPU Architecture**:
    - Reduce GPU driver overhead:
      - Reduce draw calls with a small no. of vertices: Do *Draw Call Batching*
      - Reduce GPU state changes due to a lot of different materials
    - Reduce expensive calculations in shaders:
      - Reduce materials with normal, height & occlusion maps that involve per extra pixel calculation.
      - Reduce post process image effects such as antialiasing, depth of field, etc.
      - Reduce complexity in custom shader programs
        - Reduce expensive functions such as sqrt, exp, pow, sin, cos, tan.
        - Reduce variable precision from float to half to fixed
        - Avoid branching with if statement
    - Switch back to simpler but faster legacy shaders
  - **Game Engine Architecture**:
    - Reduce Dynamic Lighting & Shadowing:
      - Bake illumination into textures
      - Reduce the resolution of the cube maps in the reflection probes.
      - Avoid dynamic reflection probes with regular updates
      - Reduce the size of the shadow maps
      - Prefer hard over soft shadows
      - Replace real-time shadows by faster shadow projector
    - Reduce Updates per Frame:
      - Not all updates need 60 fps. Use Coroutines with Sleeps
    - Reduce Physics Overhead:
      - Prefer primitive colliders over mesh colliders.
      - Reduce the fixed timestep interval.
      - Wheel colliders are expensive.

- Reduce network traffic
- Reduce mesh skinning for animated characters:
  - Reduce the no. of bones
  - Reduce the no. skinned meshes, combine meshes whenever possible

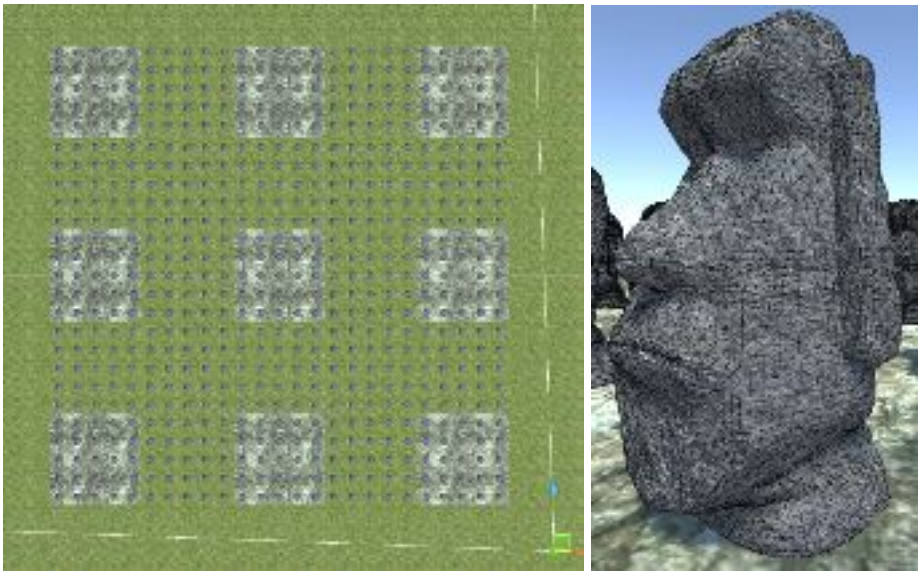
- **C# Language Architecture**:
  - Profile your code.
  - Understand the Automatic Memory Management:
    - Don’t instantiate new game objects per frame, use *Object Pooling*.
    - Prefer the creation of structs over class instances
  - Query objects or components in startup() or awake()
  - Avoid queries with string comparison
  - Remove unused callbacks such as empty update methods.
  - Know the impact of dynamic data structure
  - Reduce expensive getters such as gameObject.Transform.position.

5.4.1 Occlusion Culling

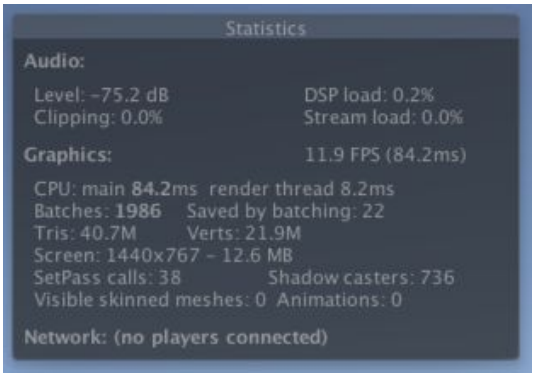
In the following two subchapter we want to optimize a large scene that is too expensive to render.

Copy the project *05 Performance Optimizations* from the DropBox folder to another folder.

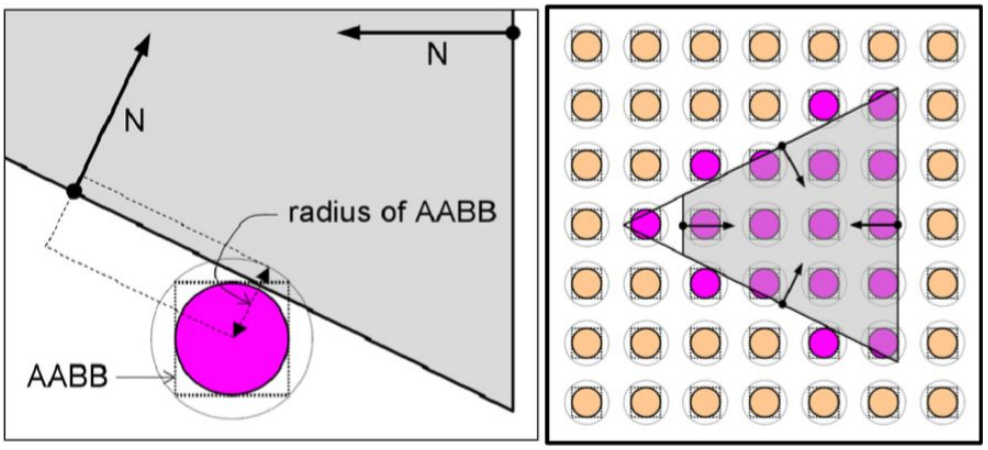
Play the scene: It contains 625 Maoi statues with 20’636 triangles each. They are standing either on the ground or on a rock.



On my *MacBook Pro (2012)* with a *NVIDIA GeForce GT 650M* I only get around 11 fps on a resolution of 1440 x 767 pixels. This is not further astonishing with 40 Mio. triangles in about 2000 draw calls:

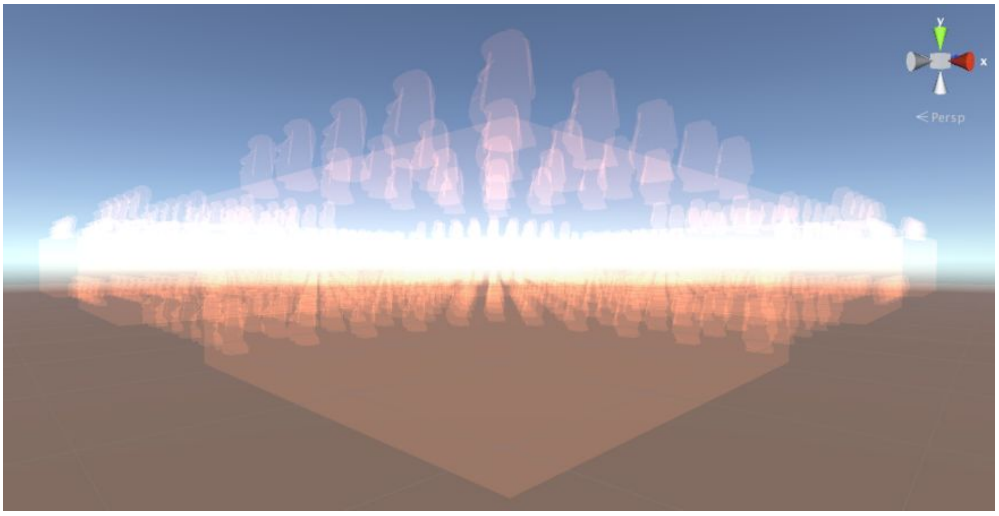


Unity automatically occludes only objects outside of the *view frustum*. Every object has to be tested against the 6 plane of the view frustum. Because testing of individual meshes would be too expensive only their axis aligned bounding box or bounding spheres are tested.



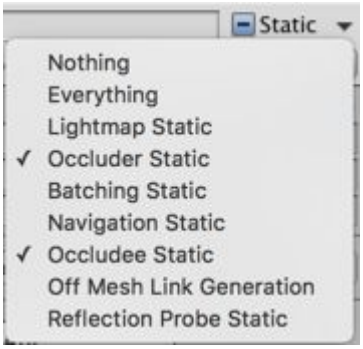
This ***View Frustum Culling*** is useless in this view angle, so that all object in the scene get drawn no matter if they are occluded or not.

A good indicator for a lot of occlusion is the ***Overdraw*** shading mode that you can set in the top left dropdown menu of the *Scene* window. The more objects get drawn over others, the brighter gets the region:

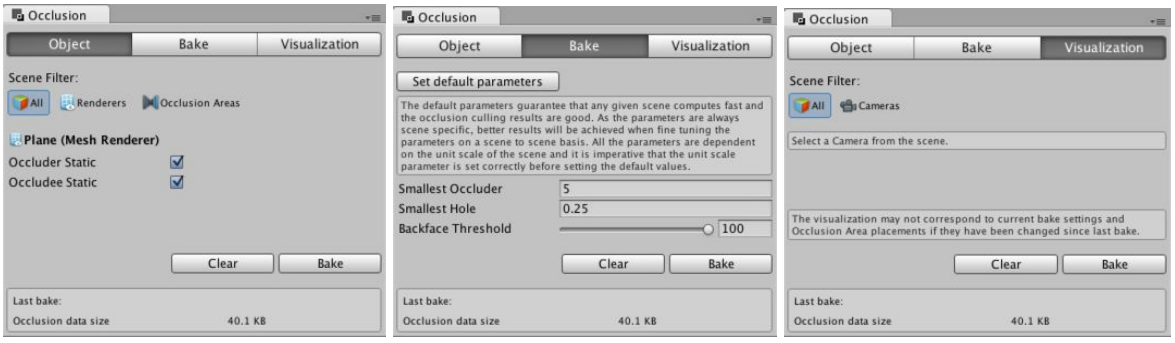


We can avoid this by using the *Occlusion Culling* included in Unity:

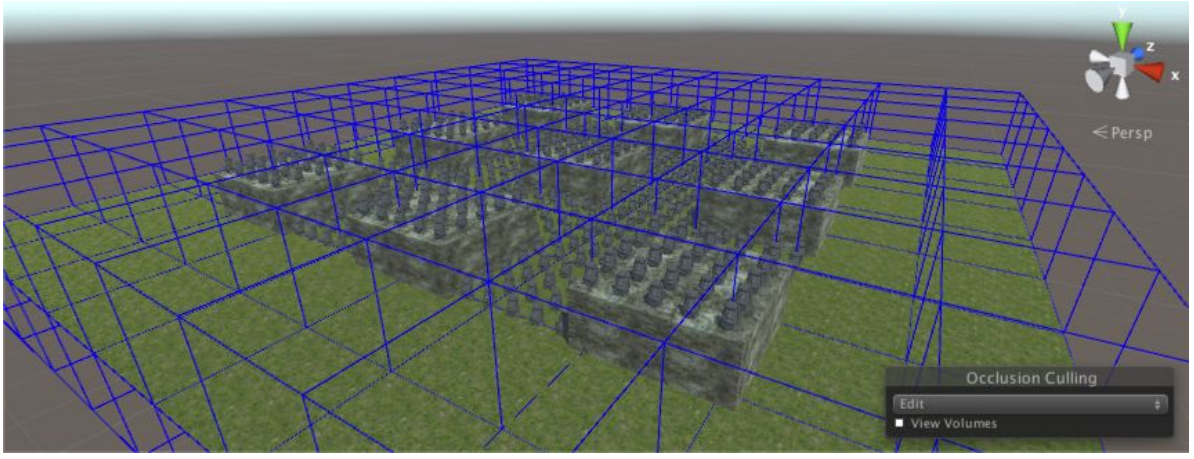
1. **Set Static**: Select the plane, the rocks and the Maoi statues and set them to *Occluder Static* and *Occludee Static*:



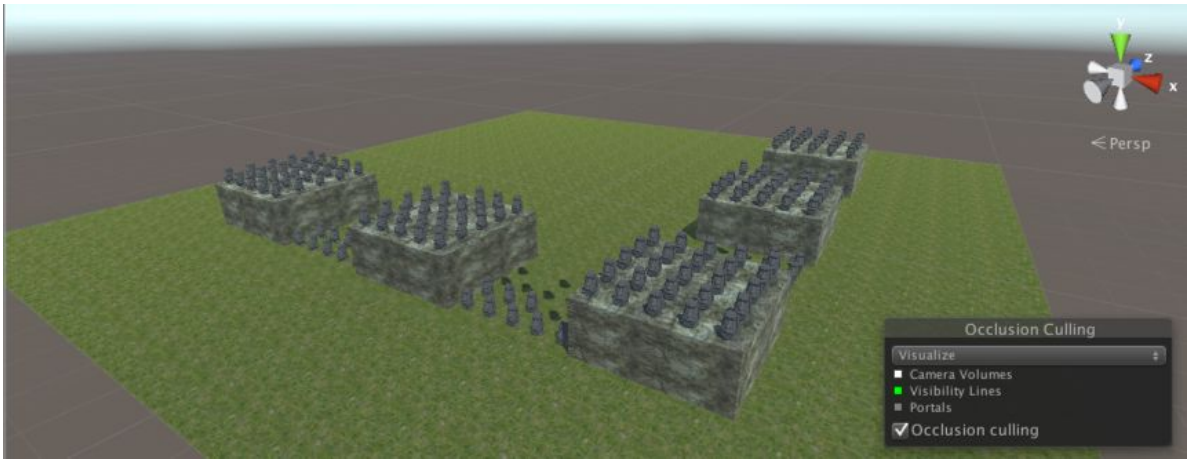
2. **Open the *Occlusion Culling*** window: It has the 3 tabs: *Object*, *Bake* & *Visualization*:



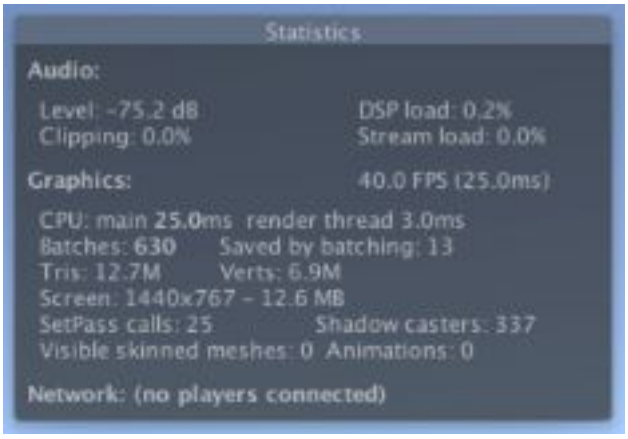
3. **Bake**: In the *Bake* tab leave all parameters and press *Bake*. An occlusion culling grid is generated depending on the size set in *Smallest Occluder* and *Smallest Hole*. From each cell A to each cell B the occlusion culling system calculates the visibility of the objects in B if the camera were in cell A. In our example only the rock will be occluders because they are larger than 5. The generation process can last a view seconds.

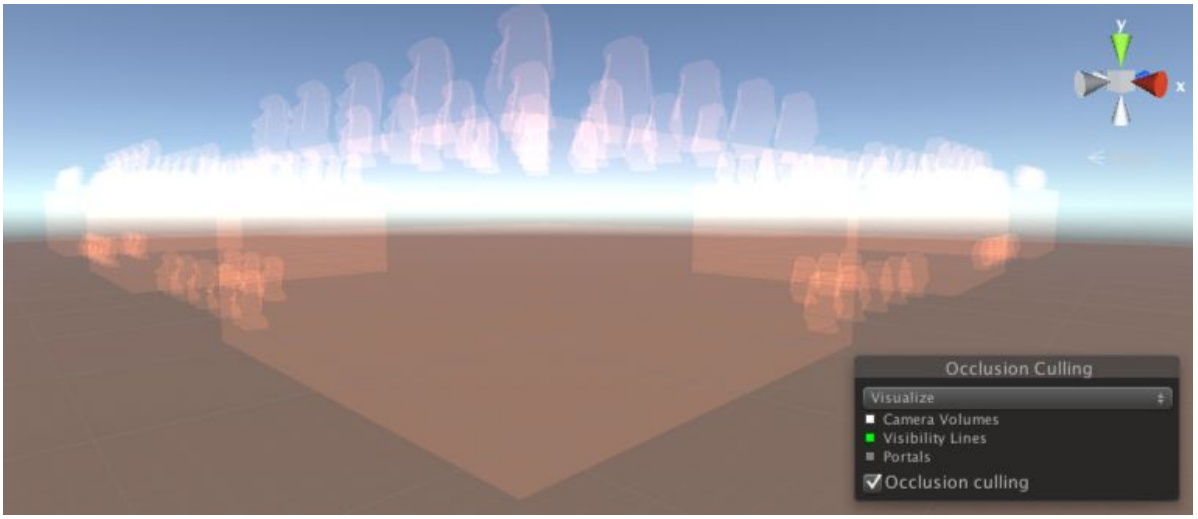


4. **Visualize the Occlusion Culling**: With the *Occlusion Culling* window open you visualize the culling effect in the *Scene* window:



5. **Play the game**: With occlusion culling the frame rate goes on my machine up to 32 fps because we only draw about 1000 objects instead of 2000:





Far less overdraw due to occlusion culling.

5.4.2 Level of Detail

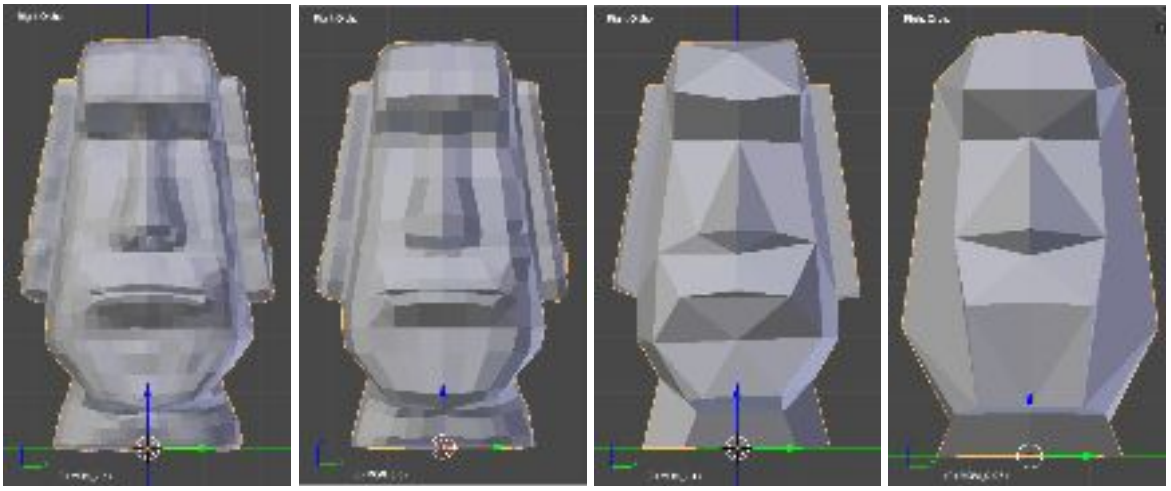
Of course occlusion culling is only a performance gain if we have a lot of occlusion in the current view. If the camera would have a bird view all 625 statues with their 20k triangles would have to be rendered and the frame rate would drop again.

In general 20k triangles in a mesh is too *highpoly* for a real-time game that should achieve 60 fps. It only makes sense in close up of the statue.

The solution for this problem is a concept called *Level of Detail* or *LOD*. Multiple meshes of the same objects with different resolutions are used. Depending on the distance to the camera a higher or lower resolution of the model is rendered.

1. **Open the *MOAO.blend* file in *Blender***: You can do this right from within Unity.

2. **Create lower resolution meshes**: We can create a lower resolution version by duplicating the mesh (*Shift-D*) in the object mode and decimating it with the *Decimate* modifier. I created 4 lower resolutions with around 2.5k faces, 0.6k faces, 0.1k, 0.05k faces and renamed the meshes accordingly. For the 2.5k and the 0.6k version I used *Un-Subdivide* method and for the 0.1k and 0.05k versions I used the *Collapse* method:



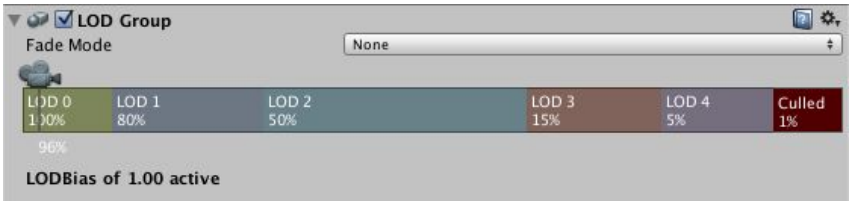
Lower resolutions with 2500, 600, 100 & 50 faces

After saving with CTRL-S the new submeshes appear also in Unity:

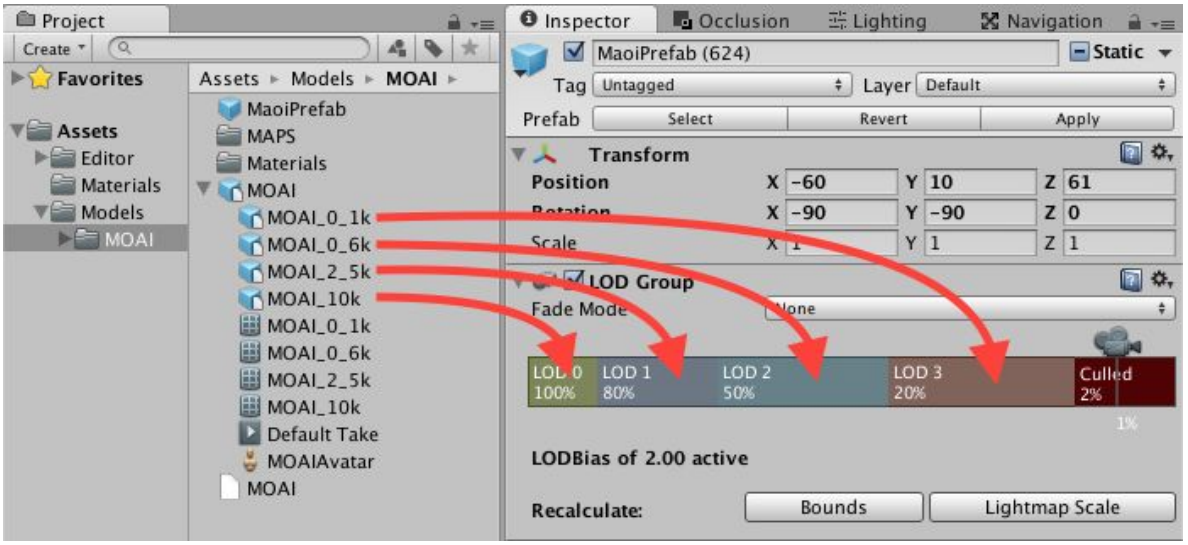
3. **Back in Unity Remove the *Mesh Filter* and *Mesh Renderer*** component from the *MaoiPrefab* prefab. Do not remove any references in the *Hierarchy* window. All statues will disappear in the *Scene* window because all references to the prefab became empty objects.

4. **Add an LOD-Group** component to the *MaoiPrefab* prefab.

5. **Add a 4th & 5th level in the LOD-Group** by right-clicking on the *Culled level*.

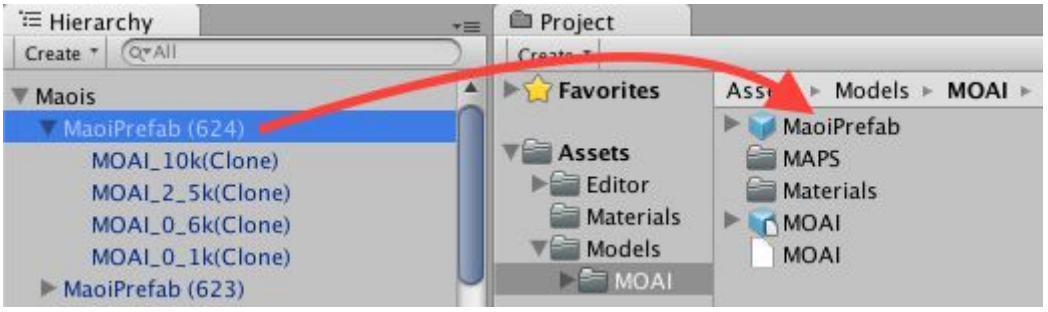


6. **Add the models to the LOD-levels**: You can not drag the models onto the levels of the prefab. You have to select first one of the prefab's references and drag the models onto its LOD-levels:



After doing this only the prefab reference will show the model. You can view the different levels by sliding the camera icon above the colored levels.
The **percentages** in the LOD bars represent the fraction of the bounding box height relative to screen height where that LOD level becomes active. You can change the percentage values by dragging the vertical lines that separate the bars.
If the ***LOD Bias*** is not 1, the camera position is not necessarily the actual position where LOD transits from one to another. You can set the *LOD Bias* in *Edit > Project Settings > Quality*.

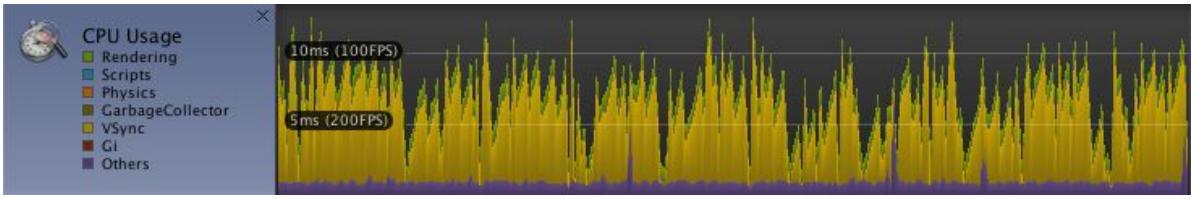
7. **Drag the Prefab reference onto the MaoiPrefab prefab** to redefine it. After this all references in the scene are updated and work with the LOD system.



8. **Play the scene!** Now we suddenly have over 80 fps and this also in the *Scene* window during editing.

In the *Profiler* window we now see a yellow ***VSync*** curve that became dominant. This is good because it shows that Unity is waiting for the vertical sync signal of the monitor to

refresh the frame. Most laptop monitors have a 60Hz refresh rate and Apple has decided to update the current frame only in between two frames.



On Windows PCs with more configuration possibilities you can turn this feature on or off. The downside of a turned off VSync is the tearing effect where you see the difference of the two subsequent frames.

5.4.3 Batching

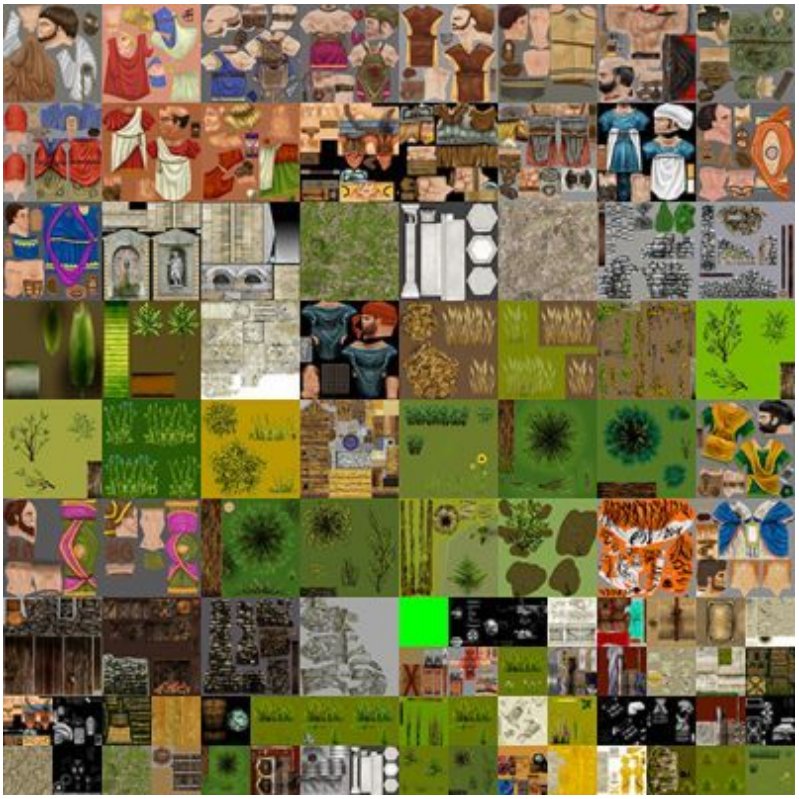
The success in the previous chapter was not only due to the savings with LOD. In the *Statistics* window you see that of the remaining 638 batches 402 could have been saved by bachting. What does this mean?

Draw Calls: 3D APIs such as *OpenGL* or *Direct3D* must receive a so called draw call. In OpenGL e.g. these are all the commands that begin with *glDraw** such as *glDrawElements*.

State Changes: Before this draw call is executed you have to specify with other commands which vertex attributes (positions, normals, colors, tangents, etc.) and shader programs with all their parameters should be used. The sum of all these state changing calls into the GPU driver can be quite high compared to the final draw call. Because of this overhead of state changes before the actual draw call we should try to do our draw calls for as much vertex data at once as possible.

Draw many Objects with a single Material with the same Textures: If we summarize the state settings as the *material* this means that we should try to draw as many objects as possible that have the same material. Because textures are essential parts of a material, we can also say that we should draw as many objects as possible with the same texture.

Texture Atlas: To allow multiple objects to be drawn with the same material we can combine their textures into one big *texture atlas*. Of course this has to be done in the modelling tools and most of them provide this functionality (see e.g. an extension for Blender).



Many textures assembled together in one big texture atlas

Unity's Static and Dynamic Batching:

- **Dynamic Batching** allows Unity to batch moving objects with the same material under the following conditions:
  - Meshes must have less than 900 vertices.
  - Game objects must have the same scale.
  - Game object that don't receive shadows
- **Static Batching** allows the engine to reduce draw calls for geometry of any size (provided it does not move and shares the same material). Static batching is more efficient than dynamic batching, but it uses more memory.
  You need explicitly specify that certain objects are *Static* or *Batching Static.*
  Author's comments:
  1) In the unoptimized *Maoi* scene no batching happened after flagging the status as static. I guess there is still an upper limit and that the 20k triangles in one statue is above this limit.
  2) Also flagging the statues as *Occluder* and *Occludee Static* allowed Unity to batch the meshes of the smaller levels.

Checkpoint 9: Performance Optimizations

5.4.4 Script Optimization

The optimization of Unity scripts needs

- Good knowhow on the C# language and its specific *Memory Management.*
- Specific knowhow about some expensive Unity methods.

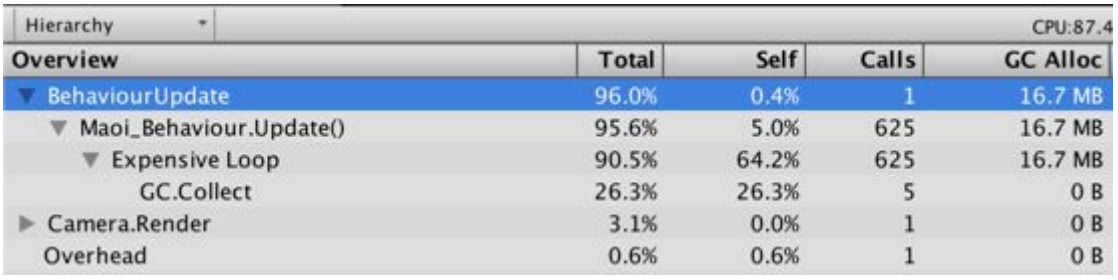
1. Add the following code in a new C#-script to a *Maoi* object and name it *Maoi_Behaviour.cs*:

```
using UnityEngine;
using System.Collections;

public class Maoi_Behaviour : MonoBehaviour {
    class myClass
    {   public float x, y, z;
    }

    // Update is called once per frame
    void Update () {
        Transform lookAt = GameObject.Find("Main Camera").transform;
        transform.LookAt(new Vector3(lookAt.position.x, transform.position.y, lookAt.position.z));
        transform.Rotate(270,90,0);

        Profiler.BeginSample("Expensive Loop");
        for (int i=0; i<1000; i++)
        {   myClass a = new myClass();
            a.x = 1.0f;
            a.y = 1.0f;
            a.z = 1.0f;
        }
        Profiler.EndSample();
    }
}
```

2. **Press *Apply*** in the top section of the Inspector of the Maoi object to apply the script to all status in the scene.
3. **Run the game**: Unfortunately the performance is bad again.

5.4.4.1 Use the Profiler to find Bottlenecks

With the *Unity Profiler* you can drill down into the methods called during the game play. The following screenshot shows that our *Update* method burns 96% of our CPU resources:



Hierarchy — CPU:87.4

Overview	Total	Self	Calls	GC Alloc
▼ BehaviourUpdate	96.0%	0.4%	1	16.7 MB
▼ Maoi_Behaviour.Update()	95.6%	5.0%	625	16.7 MB
▼ Expensive Loop	90.5%	64.2%	625	16.7 MB
GC.Collect	26.3%	26.3%	5	0 B
▶ Camera.Render	3.1%	0.0%	1	0 B
Overhead	0.6%	0.6%	1	0 B

We see that the profiler lists all called subprocedures including the ones from Unity.

If you want to profile a specific code section you have to extract it into a separate method or surround it with:

```
Profiler.BeginSample("Expensive Section");
// some expensive code …
Profiler.EndSample();
```

5.4.4.2 Know the C# Memory Management

5.4.4.2.1 Garbage Collection

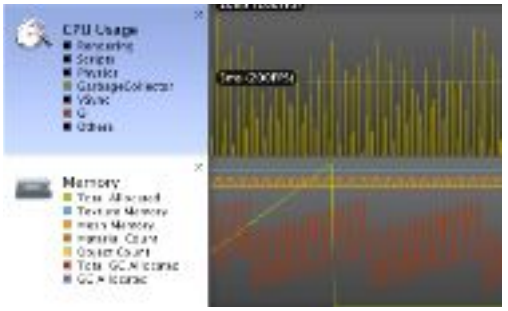
The C# language uses a managed memory model with a garbage collector (GC). As in Java every new instance of a class is created on the heap and a reference is returned. You never have to bother about the memory. The GC frees the memory periodically for you when a reference is not used anymore.

Structs: Similar to C and C++, C# also offers us structs to use as an alternative to classes. In C# a *struct* is a *value type* which means that it is passed as a value and not as a reference. Another consequence is that structs are allocated on the *stack* and therefore deallocated at the end of a block. This can have a huge impact on the performance.

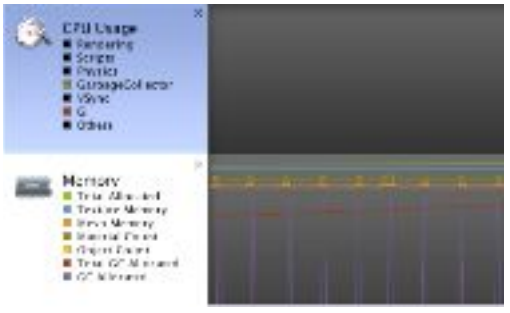
Compare the following script versions attached to the 625 Maoi game objects.

The first version uses a class object within a loop and the second version uses a struct type instead. The first runs at **7 fps** and second at **70 fps**:

```
public class MaoiBehaviour : MonoBehaviour
{   class myClass {public float x, y, z;}
    void Update ()
    {   for (int i=0; i<1000; i++)
        {   myClass a = new myClass();
            a.x = 1.0f;
            a.y = 1.0f;
            a.z = 1.0f;
        }
    }
}
```



```
public class MaoiBehaviour : MonoBehaviour
{   struct myStruct {public float x, y, z;}
    void Update ()
    {   for (int i=0; i<1000; i++)
        {   myStruct a = new myStruct();
            a.x = 1.0f;
            a.y = 1.0f;
            a.z = 1.0f;
        }
    }
}
```



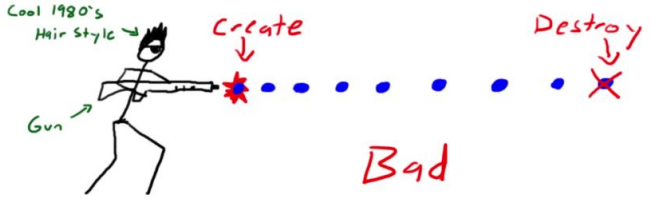
So be aware of the garbage collector and check the profiler for the GC spikes. Try to reduce memory allocations in frequent routines such as Update().

5.4.4.2.2 Object Pooling

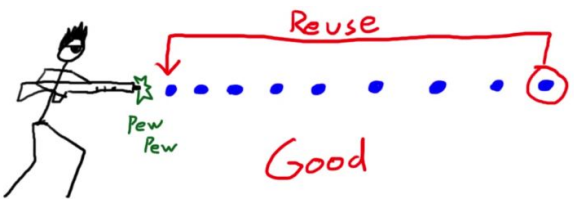
Another situation where constant object construction and the GC's destruction will have a negative performance impact is massive object instantiation at runtime with

```
GameObject obj = (GameObject)Instantiate(prefab);
```

This is often used for firing bullets at a high frequency and will lead to the GC kicking in:



The solution to avoid this problem is *Object Pooling*. The basic idea is to have a class that will store a list of game object. You will instantiate a list of objects in advance and reuse them again and again:



A pool system class could look like this:

In the constructor we instantiate a number of game objects and add them to a list:

```
List<GameObject>list;

public PoolSystem(int size, GameObject prefab){
    list = new List<GameObject>();
    for(int i = 0 ; i < size; i++){
        GameObject obj = (GameObject)Instantiate(prefab);
        list.Add(obj);
    }
}
```

During the game we get an object by removing just the first one. Like this we don't have to search the list for free ones:

```
public GameObject GetObject() {
    if(list.Count > 0){
        GameObject obj = list[0];
        obj.RemoveAt(0);
        return obj;
    }
    return null;
}
```

If we don't use the object anymore we just put it back into the list. We do NOT destroy it:

```
public void SetObjectBackInPool(GameObject obj) {
    list.Add(obj);
    obj.SetActive(false);
}
```

Of course at the end of a game or the scene we have to destroy the pool:

```
public void ClearPool() {
    for(int i = list.Count - 1; i > 0; i--){
        GameObject obj = list.RemoveAt(i);
        Destroy(obj);
    }
    list = null;
}
```

5.4.4.3 Unity Specific Script Optimizations

- **Query Game Object, Components and some Properties in advance**:
  - Query objects and components in advance in *Startup* or *Awake*.
  - Query with string comparison are slower than hash-key or tag comparison.
  - Even accessing the transform of an object is faster when a reference is cashed.
  - *Transform.position* looks like a simple C# getter property. But it is a function that iterates the entire transform hierarchy upwards to calculate the global position of the object.

In the following example we let all 625 Maoi statues look at the camera. We let them rotate only around the y-axis:

```
public class MaoiBehaviour : MonoBehaviour {

   void Update () {
        Transform lookAt = GameObject.Find("Main Camera").transform;
        transform.LookAt(new Vector3(lookAt.position.x,
                                     transform.position.y,
                                     lookAt.position.z));
        transform.Rotate(270,90,0);
    }
}
```

With maximum cashing it looks like this:

```
public class MaoiBehaviour : MonoBehaviour {
    private Transform _lookAt;
    private Transform _myTransform;
    private float     _myY;

    void Start () {
        _lookAt = GameObject.Find("Main Camera").transform;
        _myTransform = transform;
        _myY = _myTransform.position.y;
    }

    void Update () {
        _myTransform.LookAt(new Vector3(_lookAt.position.x,
                                        _myY,
                                        _lookAt.position.z));
        transform.Rotate(270,90,0);
    }
}
```

- **Update only when visible**: If many objects of the same type contain an Update method it called for all of them no matter if the object is visible or not. If the Update is useless on invisible objects you can implement the methods *OnBecomeVisible* and *OnBecomeInvisible* to flagging the visibility.

- **Independent Update Loop**: Not every object needs to be updated per frame. You can set up parallel running update loops with C# coroutines where you can sleep for longer intervals:

```
// Do stuff every 0.2 seconds
void Start()
{   while (true)
    {   // do something here …
        yield return new WaitForSeconds(0.2f);
    }
}

// Do updates every frame
void Update() { ... }
```

- **Delete empty Update methods**: Even if an empty method does nothing it gets called. Remove the empty templates if you do nothing in it.

5.5 Links & References

There is a lot of information spread over the entire Unity Documentation:

- Optimizing Graphics Performance
  - Draw Call Batching
  - Modelling Characters for Optimal Performance
  - Rendering Statistics Window
  - Frame Debugger
  - Optimizing Shader Load Time
- Optimizations for Mobile Platforms
  - Practical Guide for Optimizations for Mobiles
  - Profiling for Mobiles
  - Graphical Methods
  - Scripting and Game Play Optimizations
  - Rendering Optimizations
  - Optimizing Scripts
- Performance Optimization Tips and Tricks for Unity (Video)
- Profiler: Overview for Beginners

Other resources:

- Unity 5 Script Optimization Techniques (video)
- Unity 5 Occlusion Culling (video)
- Unite 2014 - Mobile performance poor man's tips and tricks (video)
- Unite 2014 - Big Android: Best Performance on the Mobile Devices (video)
- Unite 2015 - Mobile optimization techniques (video)
- Unite 2015 - How we optimized our mobile game (video)
- Unity 3.5 LOD Tutorial

6 Advanced GameDev Project

6.1 Project Ideas

List of topics for the project at the end of this course. You can of course come up with your own idea.

- Better Hair: Animation with cloth animation, Hair shader
- Advanced AI behaviours (better chasing, better searching with look arounds)
- Enhanced character controller (Falling, Standup, entering & exiting the buggy car)
- A good Zombie character controller
- A good day-night transition with correct lighting and environment cube map
- Driving AI
- IK climbing system, with handles on a wall
- Using decoys, create visible bullet holes for a shooter game
- RTS like building system
- Procedural generation (terrain, cities etc. Noise n stuff)
- Advanced IK usage for stairs and ramp walking
- Advanced networking with the new UNET framework
- Unity editor extension for
    - Simple road path tool
    - Simple mesh editing
    - City block house generator
- …

6.2 Project Description & Workload

- The workload per student should be around 20h.
- One of us will be present during the time of the lecture.
- Describe the goals of the project on an A4 PDF. Split the goals among the students.
- Send the project description until **20. November 2018** to marcus.hudritsch@bfh.ch
- Give the project a short name.
- Restrict your goals. Prefer going deeper than wider.
- **On Wednesday 21. December 2016 we will discuss together the projects.**

6.3 Deadline, Presentation & Deployment

- Deadline of the project is on **Tuesday 15. January 2019**.
- All projects are presented on the **Wednesday 16. January 2019**. The project should be presented with 2-5 slides per student and should last about 10-12 minutes:
    - ??:??: ???: *???*
- Projects that can not run in the WebPlayer have to zipped and sent to marcus.hudritsch@bfh.ch
- All projects that can be deployed with the *WebPlayer* should do that:
    - *File > Build Settings > Player Settings*:
        - Add the project name
        - Resolution: 1280 x 700 Pixel

        - Use the white Default *WebPlayer* Template.
        - Build the project.
        - Add to the HTML file after *«created with Unity»* the following:
            - Team Members: {Names of all members}
            - Features: {Special features of the project}
            - Instructions: {Simple instruction how to use the game}
    - Send the HTML file and the *unity3d* file zipped to marcus.hudritsch@bfh.ch
    - The WebPlayer projects will be uploaded to the BFH internal page: https://www.cpvrlab.ti.bfh.ch/bachelor/wm/BTI7527/proj/

6.4 Grading

- The project will count 40% to the final module grade.
- The project itself will be graded as follows:
    - 75% Quantity & quality of the implemented features
    - 25% Presentation (Language, Slides & Demo)