Final project memory

Group 7:

Marc Guerrero Palanca

Tatsiana Palikarpava

Alberto Pérez Abad

Contents:

1. Techniques used

In this chapter we describe those methods that let us create our final version of the video, involving such areas as interpolation, velocity control, orientation control, inverse kinematics and more.

Here you can find our video:

https://youtu.be/sT9KAY-xFUE

In the scene you can see the following objects: several persons, a ball, a shuriken and an arrow.

The last three ones illustrate the use of our add-on. Their trajectories were created beforehand by inserting several location keyframes. Then was applied our tool with the following properties:

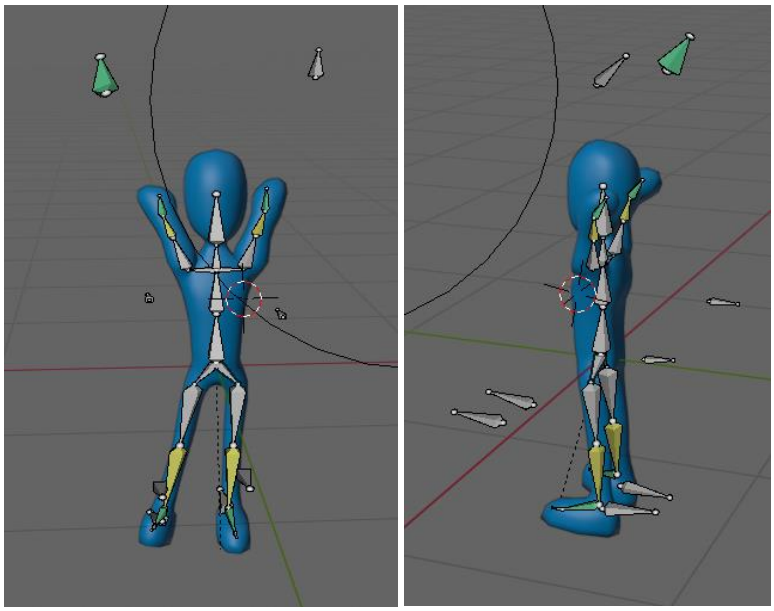
- *Ball*: Hermite interpolation, velocity control with reparametrization.
- *Shuriken*: Linear interpolation + random movement, orientation control.
- *Arrow*: Linear interpolation, orientation control.

The idea is that we want a ball to follow the trajectory with defined velocity to make its movement more natural (e.g., when it flies up to a person it slows down). This effect was reached by amending current distance property values in Graph editor. One more important issue is orientation control. With its help we obtained the effect of the arrow staying aligned with the trajectory. We also used this option for a shuriken to simulate its rotation.

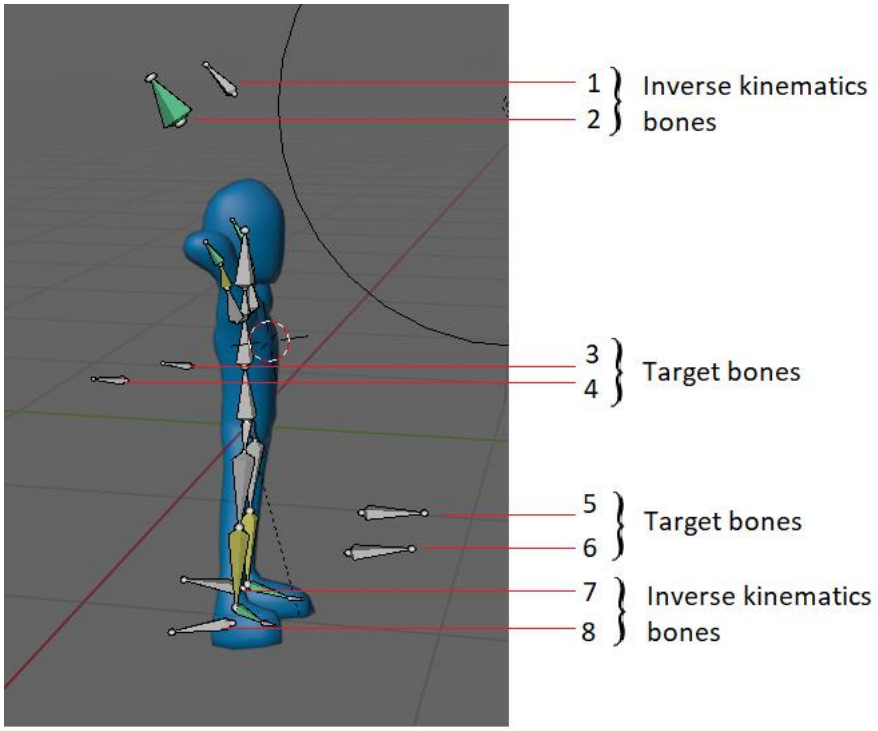
We created 2 objects of type Sun and a plane to make natural lights and shades in the scene.

We have also animated characters created with the help of MOCAP and doing some exercises. They were imported as .bvh files recorded in capture room and the geometry of the virtual actor was adjusted to the real actor. Their clothes and faces are designed using Texture paint.

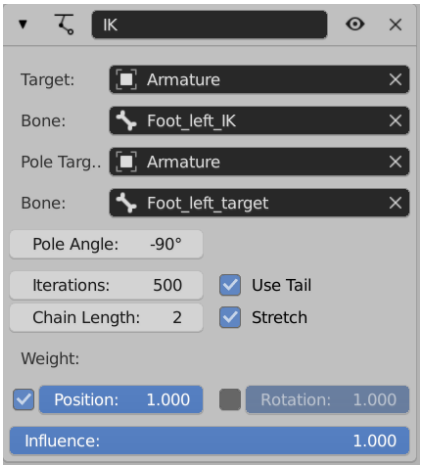
One more animated character (doing a flip) is designed by means of inverse kinematics. Below you can see the skeleton used.



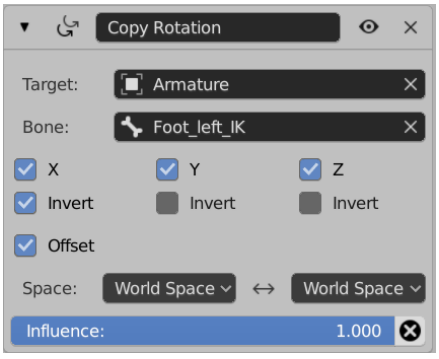
Apart from usual bones we added 8 more: 4 for legs and 4 for arms.



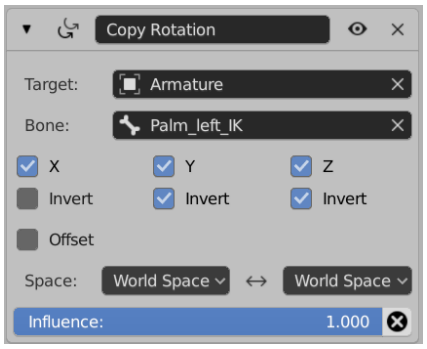
The idea is that inverse kinematics bones (1, 2, 7, 8) are disconnected from the main skeleton and used as target in inverse kinematics bone constraints.



Additionally, corresponding bones in the main skeleton are complemented with copy rotation constraint (where the rotation is copied from IK bones again).



In turn, target bones (3, 4, 5, 6) are used as pole target in inverse kinematics bone constraints. They prevent elbows and knees from rigging in the wrong direction.



And one more thing that facilitates movement design is copy rotation constraint for bone 2 (it is used to avoid setting the rotation manually for symmetrical bones).

In fact, we have 6 characters doing different exercises, but for creating a video we rendered its parts separately and then joined them during video editing.

In the beginning of the video you can see the title appearing from the wood. Here we describe briefly how to do it.

First of all you need to create a text and give it the form you want. Then select the text, add a remesh modificator and set the octree depth value to as many pieces as we want it to be divided. Then we transform the text into a mesh in order to set image as a texture.

Next we create a displacement vector node and link the displacement output to the displacement input of the material output node and the output color of the image vector to the height input of the displacement node. Then we proceed to edit the cube where the text is contained where we can set it as we prefer. Further we add a wireframe modifier to the cube to give the inside of the cube more geometry and thickness and set the material as the one we set to the text. To create the effect we go to the object menu and select quick effects > quick explode. In the particles menu we set the particles emission to 10000 and set the frame where the effect starts and ends and the lifetime of these. On the source extension we mark the check next to "use modifier stack" so the wireframe modifier will be used. Now we have the animation where the entire cube explodes at the same time, but we want it to go from one side to the other. To do this we need to add a blend texture to the particle system, so in the particles panel we open the textures section and add a texture. We edit it into the texture panel where we select the "particle settings - texture" and select Blend as type and change the mapping coordinates to UV and create a new map; on the edit mode we unwrap the cube selecting "Project from view (Bounds)", so the cube now explodes from one side to the other. Below you can see the link to the source from which we have learned how to do this.

https://www.youtube.com/watch?v=qUAw91HPFoE&t=660s

2. Code structure

In this section we describe the structure and logics of our code while performing movement control.

Basically, it consists of two parts: the inner part (movement management) and the outer one (user interface).

User interface is provided by creating a class *MyProperties* that contains fields with properties required to be defined by user. To launch an add-on we call a function *register(),* which registers classes *MyProperties, Interpolation* and *ModifyTrayectoria. Interpolation* class is designed to display the interface of the tool (by calling *draw()* function). It also provides safer work with a tool by disabling conflicting options. *ModifyTrayectoria* class has *poll()* and *invoke()* methods. *poll()* method prevents user from invoking operator with invalid input (e.g. no object selected, no keyframes defined, etc.). *invoke()* method performs movement management.

Movement management consists of three stages: interpolation, velocity control, orientation control. Previously, if velocity control is required, we construct a table that contains distance run by the object by current frame [*construir_tabla()*]. Then we go through the frames in a loop calculating corresponding object position for a current frame [*get_pos()*].

If we deal with velocity control the frame is calculated depending on the distance expected to be covered by this time [*busca()*]. Function *get_pos()* identifies two preceding and two subsequent location keyframes that are then passed to the function *interpola_valores()* that in its turn calculates a required coordinate, so we call this function three times. *interpola_valores()* calls *lineal() / hermite() / catmull_rom()* function depending on interpolation type. If we need random movement, we call *vibrar()* that calculates an

addition to the coordinate. For Hermite interpolation we retrieve velocity values. After all this steps we have a list of coordinates for corresponding frame range. We eliminate previous animation data and insert new values as keyframes.

For orientation control we apply the following actions. We remove previous rotation data and for all frames in frame range calculate a vector of the object's movement direction. Then we calculate a quaternion rotation using *get_quat_rot().* This function calls a function *get_quat_from_vecs*() that returns a quaternion that we need to apply to rotate a corresponding axis to align it with tangent vector of the curve. Then we again apply this function to make object's lateral axis horizontal (we obtain it with *get_lat_vec()* function). Finally, we calculate the quaternion to obtain lateral tilt required by user and find a cross product of all these quaternions. After quaternion is calculated its value is stored as a rotation keyframe.

3. Additional programming

In our code we have a function *CopRuta(ini_obj, list)* which is designed to copy the trajectory of the object *ini_obj* to all the objects in *list* internally. It is the possibility of action management which we used to compare the effects of different interpolation methods.
We have also implemented the function *slerp(q1, q2, u)* to get the quaternion by applying SLERP interpolation with factor *u*. We designed it to see whether we would obtain the same effect with built-in SLERP function, the effect was just like with blender method, so we used the internal function, but it is possible to replace it.

4. Changes in the code

While working on final project we have changed the following things in the code:

- As we apply interpolation, velocity and orientation control for a certain frame range, it is necessary to decide which frames are chosen as *start* and *end.* Earlier these frames were the first and the last frames of scene timeline, but in case of having the first and the last keyframes not in the same moment as the scene timeline starts and ends it was producing some problems (in particular, the object continued its movement after its last keyframe). Now it is fixed so that *start* and *end* are defined as first and last keyframes of the object's movement. In the case of extending timeline *end* frame is the one calculated according to the length of trajectory.

5. Annex: Manual

User manual is provided in a separate file *Manual.pdf*.