Examples Guide

1 EXAMPLE 1 - INDEPENDENT PARTICLES: HIGH-LEVEL TRANSFORMATIONS

This is the simplest example in this document to show how direct manipulations can be easily achieved in ***OpenMPD***. This example shows how to control an independent particle in the scene by direct transformations through the Unity UI, by simply clicking (selecting) and dragging (moving or rotating) the particle or group of particles in the scene. This can be achieved by using the Unity UI tools usually on the top left (translate and rotate, see Figure 1.A) or by manually changing the translation and rotation values per axis on the transform section inside the inspector tab as shown in Figure 1.B (at least one GameObject/Particle must be selected on this action). This particular example takes advantage of Unity's update rate by wrapping up the transformation matrices from each active primitive in the scene and committing the update to OpenMPD once per Unity update call (~300ups). We also call this update type "high-level" updates.

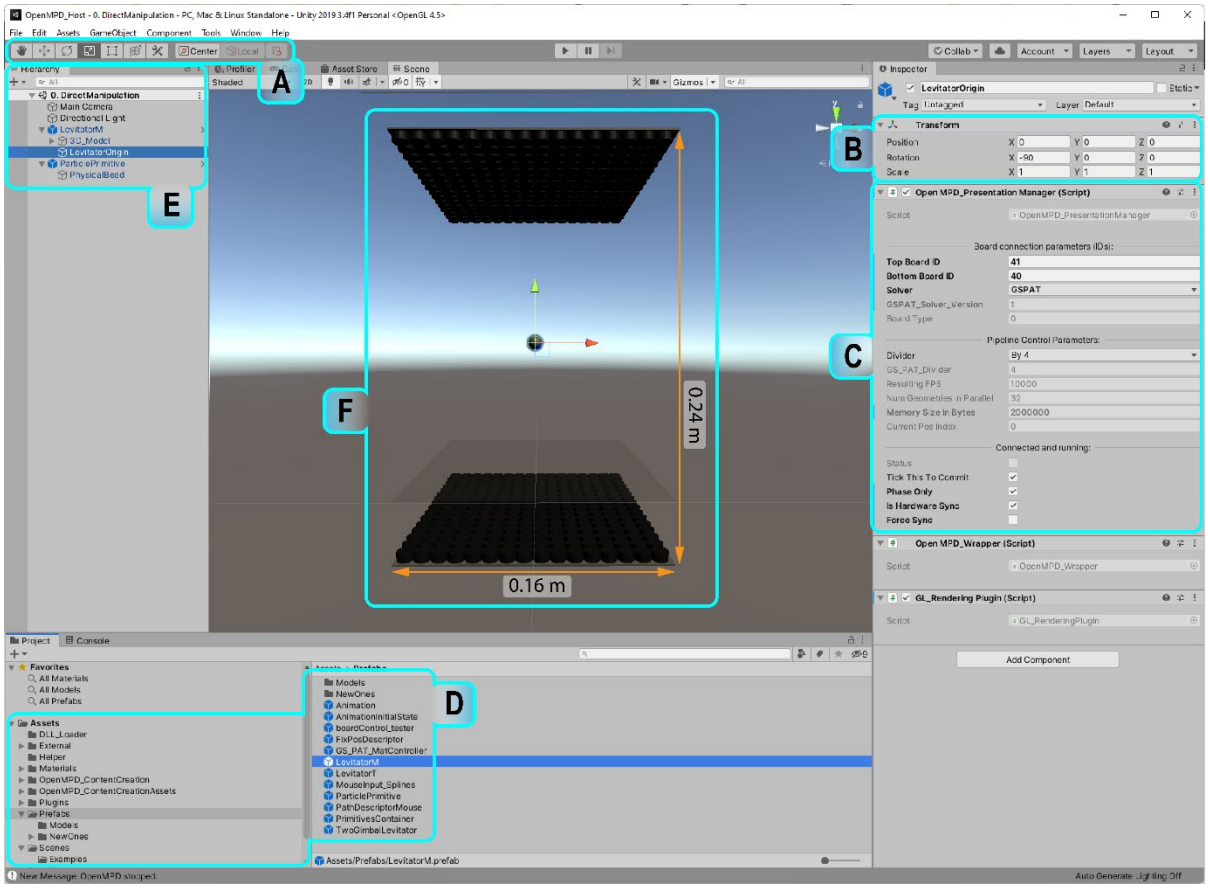

Figure 1: Example 1. Direct manipulation example.

For this example we need to open up a clean Unity scene and follow the steps listed below:

1. **Add and configure the Levitator prefab**: The Levitator prefab can be found in the Prefabs folder (Figure 1.D), drag and drop it into your scene. Locate it on the "Hierarchy" tab and click the dark grey triangle on its left to reveal the inner components, then select LevitatorOrigin and focus on the *OpenMPD_PresentationManager*. Once there, configure board ID fields (see section 2.1.1 in "OpenMPD Tutorials_Guide" and its Figure 1.B) and solver parameters (see "OpenMPD Tutorials_Guide" section 2.1.2 "Important note" and its Figure 1.C). In this example we used 41, 40, and GSPAT for IDs and solver respectively, leaving the other fields as in Figure 1.C. However, if you don't have a MPD device connected, please set both IDs to 0 to enter the simulator mode. It is important to note that each time you drop a GameObject (e.g., prefabs, default geometries or empty objects) into the scene, it may be randomly located, please make sure you reset its location to (0,0,0), before doing anything else.

2. **Drag and drop into the scene a ParticlePrimitive prefab**: Set its location to be more than 3cm away from the bottom or top boards of the levitator, as conventional setup uses 0.16m (in X), 0.24m (in Y) and 0.16m (in

Z) as shown in the Unity virtual model (Figure 1.F), we set it at the central position (0,0.12,0). This particle has a *OpenMPD* Primitive module attached to it with two parameters MaxStepInMeters and MaxRotInDegrees (with default values of 0.00025 and 1 respectively). They both are variables to constraint the maximum step used to translate and rotate a particle per update, to ensure a smooth particle transition during the animation. These two variables can be tuned base on your animation and setup needs, however, for the examples in this document, it is best to keep the default values.

3. **Use and MPD device or simulator mode:** If you have a MPD device, this is the point where you need to make sure it is connected (USBs correctly recognized) and powered
4. **Press play**: Once you press play, you would be able to visualize a static levitated particle in the centre of the virtual space in Unity, if the levitator is connected and correctly configured, you would be able to place a physical particle at the centre of the physical levitator or in the location you defined the particle to be in the Unity UI. You can use some twicers to position the physical particle in the correct location.
5. **Manipulate the particle**: Once the physical particle is in place, please select the translate icon on the top left area in the Unity window (see Figure 1.A), then just left click and select the virtual particle primitive in the Hierarchy tab (Figure 1.E). Then, the particle on the scene tab would be highlighted in green and a coloured arrows will appear (one per axis, see (Figure 1.F), and this indicates that we can now directly move the particle around the levitator's volume by dragging each arrow independently (to move along a single axis) or all together. At this point please observe that the physical particle matches the displacement from the virtual particle.

It is worth noting that the Primitive component (*Primitive.cs*) is directly sending a transformation matrix (composed by current position, rotation and scale) per primitive in the scene to the OpenMPD_ PresentationManager once per Unity update call (~300 ups), we also called high-level updates.

Exercise A: *Manipulating multiple particles*. Now you can try adding another particle to the scene and move them around one at a time. Try to test the limits of your rendering space (what is the best rendering volume size for your device?) and how close can the particles be on each axis (what is the minimum between-particle distance?).

Exercise B: *Manipulate two particles as a group*. To do this, add an empty *gameobject* to the scene (set it to the centre), and make the particles children of this new *GameObject (drag and drop the particles into it)*, then any translation or rotation on the parent will affect the children. Here is a good time to test rotations per axis, are they behaving equally across axes? What is the main difference? How is the particle's displacement behaving? Other parameters you can test are *MaxStepInMeters* and *MaxRotInDegrees,* are the default values best performing for your device?

2 EXAMPLE 2 - SINGLE PARTICLE: FAST-MOVING PARTICLES (POV)

In this example, we will use a single primitive to generate a circle animation (by accelerating a particle in a circular path, see Figure 2). Opposite to the Example 1 - Independent Particles: High-Level transformations, where the particle displacement was set through Unity transformations and constrained by its update rate (high-level updates ~300ups), in this example we will use low-level descriptors to define the particle trajectory and velocity we need, allowing update rates up to 10k ups. In this example, we will first open an empty scene, then the steps that we will follow for this example are:

1. **Add and configure the Levitator prefab**: The Levitator prefab can be found in the Prefabs folder (Figure 1.D), drag and drop it into your scene. Once there, follow the instructions in Example 1 - Independent Particles: High-Level transformations, Step 1.

2. **Add some GameObjects to structure our content:** First, add a "Contents" prefab from the prefabs folder. This will serve as a container for the primitives in the scene. It also contains a script that enables unity to simulate particles movements in run-time defined through descriptors (*UpdateParticleAnimation.cs*). Make sure you set the Contents prefab's location to the centre of the levitation space, by literature we assume a standard levitation volume with sizes 0.16m (in X), 0.24m (in Y) and 0.16m (in Z), then the centre would be at (0, 0.12f, 0). Now let's add two additional empty GameObjects and rename them as "Descriptors" and "States" respectably. Same as Contents, these new elements will be the containers of the new descriptors and states we will use to define and control our animation. As these new containers' location will not affect the particles' performance directly (just the "Contents" container influences particle performance in high-level), then we can set the location for "Descriptors" and "States" to be (0,0,0).

3. **Add a primitive prefab as child of the Contents node**: Once there, follow the instructions in Example 1 - Independent Particles: High-Level transformations, Step 2.

4. **Add an empty GameObject as a child of the Descriptors node**: First, change the GameObject's name to "FixPosDescriptor" and attach a *FixPosDescriptor.cs* script to it (see ***CodeExample 1***), there are two methods to do this. i) Manually: inside the Assets folder on the project tab, go to "Open*MPD_ContentCreationAssets/ OpenMPD_DescriptorAssets/* " and drag and drop the "*FixPosDescriptor.cs*" file into your renamed GameObject. Or ii) UI-based: press the button "add additional component" on the Unity inspector when your GameObject is selected, and enter the script name you would like to attach to it, in this case, enter "*FixPosDescriptor*" and select it from the drop-down list. Once the descriptor is attached to the GameObject you will have some fields (x, y, z) to enter a static 3D position that can be used for animations or transitions (Figure 2.F). This descriptor usually serves as the initial state for your animation. For this example, set the (x, y, z) position as (0.02,0,0), remember that the units are defined in meters so 0.02m represents 2cm. To avoid possible failures during the rendering stage, keep in mind that the available working volume of the device is 0.16m in x, 0.24m in y and 0.16m in z, (to date there is no constraint implemented to warn you when contents leave this volume, be a bit careful), but the recommended rendering/working volume size is 0.1m x0.1m x0.1m (right in the centre if the levitation space).

Important Note: The position defined in the *FixPosDescriptor* is not the position of the virtual particle visible in the Unity UI (GameObject>Inspector>Transform), but the position where the physical particle will be in the real world when animation is on, and the state that holds such descriptor is enabled. In other words, the descriptors define the positions where the physical particle will go through once in play mode.

We can visualize the structure composed of primitives, descriptors and states as a state machine, where we have two main states, Current, and Next. We can define both states by using *DescriptorAssets*.cs component per state and switching between states through high-level mechanisms such as scripts, events, collisions or using an editor checkbox in Unity. During the state transitioning the framework disables the descriptors currently enabled per primitive and enables the descriptors associated with the primitives present in the next state.

Important Note: The transition between states must meet a continuity condition per particle, this means that the starting and final position on each trajectory defined by the descriptor per particle must be the same to enable

path looping. Also, the final position from the current state must match with the initial position of the next state, to avoid rendering issues (losing the particle during state transitions).

CodeExample 1

```
public class FixedPositionDescriptor : PositionDescriptorAsset
{
    [Header("Position:")]
    public Vector3 position;
    public bool updateDescriptor = false;
    [ReadOnly] public uint desciptorID = 0;

    float[] positions;// = new float[4];

    void Start(){
        updateDescriptor = true;
    }

    // Update is called once per frame
    void Update(){
        // Update when OpenMPD is correctly instantiated & update is requested
        if (updateDescriptor && OpemMPD_PresentationManager.Instance()){
            // Populate the positions buffer
            GeneratePositions();
            updateDescriptor = false;
        }
    }

    void GeneratePositions(){
        // Generate the array of positions from the position fields
        positions = new float[4] { position.x, position.y, position.z, 1 };
        // Define the position descriptor
        descriptor = new PBD_Positions_Descriptor(positions);
        // Update the descriptor id
        desciptorID = GetDescriptorID();
    }
}
```

CodeExample 1 shows the code implemented on the *FixedPositionDescriptor* script. All the example code we will go through in this tutorial will present the same element's structure in the implementation code, starting at the top with i) the class name and inheritance; in this particular example the class name is "FixedPositionDescriptor" and we define that this class is inherited from the "*PositionDescriptorAsset*", this allows the class the access to the OpenMPD framework resources, for instance, update arrays of positions to the device in runtime, retrieve rendering states and other features that we will explain along with these tutorials. ii) The variables definition; these can be private, protected or public, however, it is important to remember that in Unity public variables are not only accessible from other scripts and classes, but also they are a link to allow a dynamic definition of values in run time from the Unity UI on the inspector tab, iii) native Unity methods like Start(), Update(), etc., and finally, iv) utility functions we defined to make our requested computations.

FixedPositionDescriptor is a basic component that only takes position in a Vector3 form (x, y and z fields) to generate a single-position descriptor (see "GeneratePositions" function in the **CodeExample 1** box).

5. **Add a second empty GameObject as child of the *Descriptors* node**: first, change the GameObject name to "CirclePosDescriptor" and attach the *circlePosDescriptor.cs* script to it from *OpenMPD_contentCreationAssets/OpenMPD_DescriptorAssets/CirclePosDescriptor.cs* (drag and drop into your renamed gameObject, see Figure 2.A) or just press the button "add additional component" on the Unity inspector when your GameObject is selected and enter the script name you would like to attach to it. This script is an example of a position descriptor that generates the positions around a circular path allowing users to define the radius of the circle and the number of samples in the path (see Figure 2.B). For this example,

use a radius of 0.02m and a sampling number of 5000. Using 5000 samples and a rendering framerate of 10000 means that the particle will spin 2 times per second around the circle.

CodeExample 2

```
public class CircleDescriptor : PositionDescriptorAsset
{
    [Header("Circle parameters:")]
    public float radius = 0.02f;
    public uint numSamples = 1024;

    [Header("Update Descriptor")]
    public bool updateDescriptor = false;
    [ReadOnly] public uint descriptorID = 0;
    [ReadOnly] public Vector3 initialPos;

    // Local variables
    float[] positions;

    void Start()  {
        updateDescriptor = true;
    }
    // Update when OpenMPD is correctly instantiated & update is requested
    void Update() {
        if (updateDescriptor && PBD_RenderingManager.Instance()) {
            GeneratePosArray();
            updateDescriptor = false;
        }
    }
    void GeneratePosArray() {
        //Determine how many smaples per second we will need:
        positions = new float[4 * numSamples];
        //Fill circle:
        for (int s = 0; s < numSamples; s++) {
            float angle = 2 * (Mathf.PI * s) / numSamples;
            positions[4 * s + 0] = radius * Mathf.Cos(angle);
            positions[4 * s + 1] = 0;
            positions[4 * s + 2] = radius * Mathf.Sin(angle);
            positions[4 * s + 3] = 1;
        }
        // update the initialPos to be visible in hte UI
        initialPos = new Vector3(positions[0], positions[1], positions[2]);
        //Create descriptor
        descriptor = new PBD_Positions_Descriptor(positions);
        // retreive desciptor id
        descriptorID = descriptor.positionsDescriptorID;
    }
}
```

Similar to the *CodeExample 1*, the *CodeExample 2* also follows the same element's structure. However, the implementation for the *CircleDescriptor.cs* script not only passes the input location values from the inspector to the descriptor but uses the inputs (origin position, radius and the sample size) to generate a path from the circle equation with a specific length (see "*GeneratePosArray"* function on the *CodeExample 2*).

6. **Add a two empty GameObject as child of the States node**: Rename them as "Idle" and "Animation", then attach a *PrimitiveStateAssets,cs and a StateSwitch.cs* script to each of them from *OpenMPD_contentCreationAssets/ OpenMPD_PrimitiveStateAsset folder* (Figure 2.A)**.** These new GameObjects can now hold the primitive's parameters such as amplitudes, phases and positions to be used at each state in our animation rendering and across animations. The component *PrimitiveStateAsset* defines the particle's parameters per state. While *StateSwitch* component (Figure 2.E) will be in charge of handling

the framework resources during the transition between states. In this example “Idle” state holds the initial set of parameters to define the initial state, in other words, this will define where is the initial position for the particle before starting the animation (a buffer with a single element = a static position) and for now we will keep the default amplitude value (~20000 Pa).

Important Note: OpenMPD support the use of position, amplitude and colour buffers. Each primitive must have associated at least one buffer of each type per primitive, which represent potential configurations and behaviours. *OpenMPD Primitives* are free to declare as many buffers as required (i.e., to enable changing states to present different shapes, different haptic patterns, or audio), however, only one set of buffers (position, amplitude and colour) will be active per primitive at each point in time. If during run-time one of the buffers is missing, the default values will be used by OpenMPD.

To configure each state (“Idle” or ”Animation”) in this example, we need to pass: **i)** one primitive, **ii)** one position descriptor, **iii)** one amplitude descriptor and **iv)** one colour descriptor (see Figure 2.D). For instance, click on the “Animation” state in the Unity Hierarchy tab to visualize the scripts on the inspector tab, drag and drop the particle primitive we have in the scene into the Primitive field, and similarly drag and drop the CirclePosDescritor object from the Hierarchy tab to the PositionDescriptor field in the inspector tab (see Figure 2.D). Leave the amplitude and colour descriptors empty for this example and the framework will use the default values for these buffers. Leave the startingPositionSample, startingAmplitudSample and startingColourSample in 0, they represent the index in the buffer (position, amplitude and colour buffers) to be assigned as starting index for the animation.

Similar to the “Animation” state, the “Idle” state holds the descriptors for the initial set of buffers. Repeat the procedure from the “Animation” state but instead of using the CircleDescriptor on the PositionDescriptor field drag and drop the FixPosDescriptor GameObject and leave the other fields with the default values as shown in Figure 2.D.

Important Note: you can define as many states as you need, however only one will be active in play mode, then if all the states are active when pressing the play button, OpenMPD will go through all of them and leave only the last one active. Please leave only the initial state active (disable all the animation states in the scene by disabling the game object to which they are attached). In this example, the only state enabled is Idle, allowing us to place the particle in a static position after pressing play.

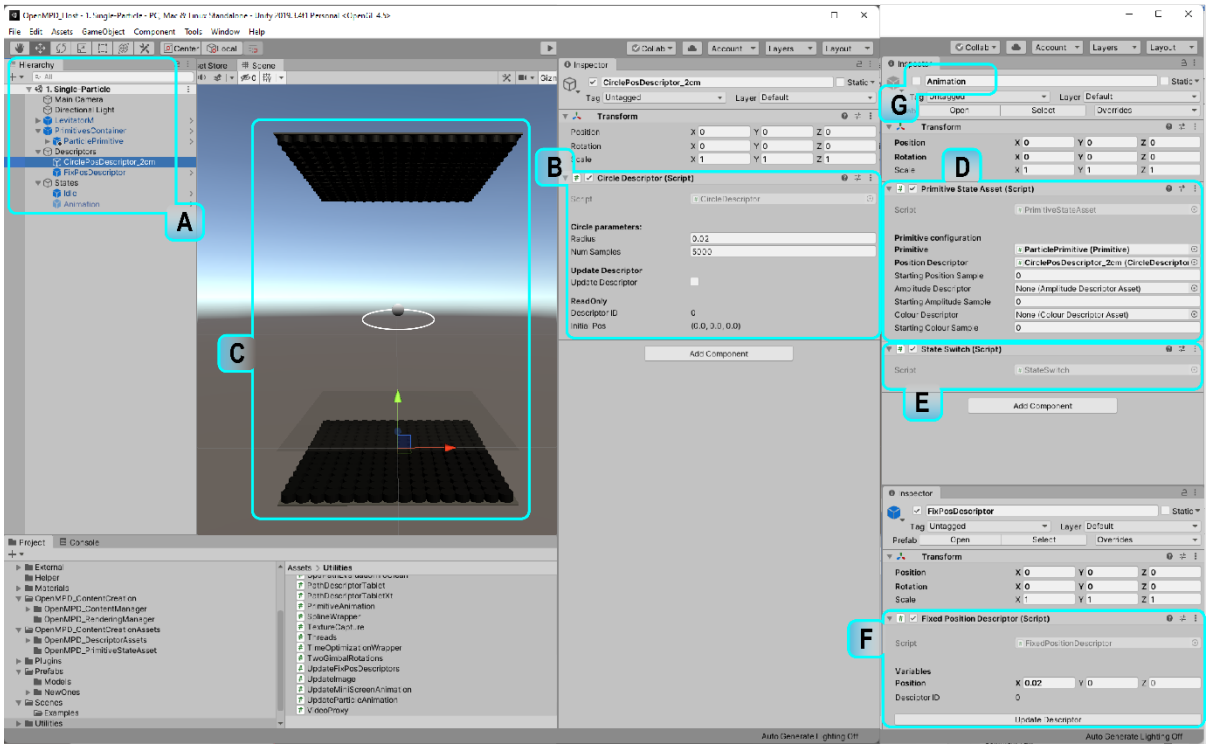

Figure 2: Example 2. Single particle in a circular movement.

7. **Use a MPD device or simulator mode:** If you have a MPD device, this is the point where you need to make sure it is connected (USBs correctly recognized) and powered. If don't, then the unity visualization will reflect the indented particle behaviour when pressing play.

8. **Press play**: Once you press play, you would be able to visualize a static levitated particle in the position defined on the *FixedPosDescriptor.cs* inside the virtual space in Unity. If the levitator is connected and correctly configured, you would be able to place a physical particle at the staring location. You can use some twicers to position the physical particle in the correct location.

9. **Changing animation states:** To change between states you need to manually activate the *Animation* GameObject by activating the checkbox next to the GameObject name ("Animation" or "Idle") in the inspector tab (see Figure 2.G), and automatically the animation will switch from Idle to Animation state (and the Idle state will be then disabled) while the rendering of the animation will start. Remember that to create any new animation state must have both the *MPDPrimitivestateAsset* and *MPD_StateSwitch* scripts attached to it to work correctly and leave active only the initial state before the rendering stage.

By now the levitator (virtual and real) should be able to render the circular path with your defined radius and speed. So, this is all for now. This example introduced you to quite a few relevant bits related to using Unity structure based on primitives, descriptors and states.

Exercise C: *Exploring maximum speed displacements*. This is a perfect point to explore the capabilities of our device to accelerate a particle without making it fly away. Explore the maximum velocities by adjusting the sample size, remember that having a sample size of 2000 with a target framerate of 10000 ups will end up with the particle completing 5 rounds in a second. What are the maximum revolutions per second that your device can support?

Exercise D: *Exploring Hybrid updates*. What happens if we use direct manipulation from example 1 while the particle is drawing the circle? let's find out! This is also called hybrid updates (Combining low-level and high-level updates).

3 EXAMPLE 3 - MULTIPLE PARTICLES: INDEPENDENT ANIMATIONS

This example also follows the content-descriptor-state structure shown in Example 2 - Single Particle: Fast-moving Particles (PoV) (same scene elements). In this example we are going to use the circle animation, however, we will use two particles to draw the circle in the centre of the rendering space. This can be done by two methods;

a) *Using independent descriptors to plan the circular motion per primitive*: Duplicating the steps in example 2, once per particle. This means that each primitive will have an independent position descriptor to rule its behaviour during the rendering state.

b) *Sharing the same position descriptor*: Two different particles would use the same descriptor but adjust starting indices of each particle on the drawn path.

As the option "a" is just duplicating the steps in example 2, in this example, we will follow option "b". We will start from our empty scene. Then, please follow the steps from 1 to 6, described in example 2, and then:

1. **Add a second primitive prefab as child of the Contents node**: Once you have the scene from Example 2 - Single Particle: Fast-moving Particles (PoV), follow the instructions in Example 1 - Independent Particles: High-Level transformations, step 2, then rename it as "ParticlePrimitive2".

2. **Add an empty GameObject as a child of the Descriptors node**: First, change the GameObject's name to "FixPosDescriptor2" and attach a *FixPosDescriptor.cs* script to it (Example 2 - Single Particle: Fast-moving Particles (PoV) step 4, see Figure 3.A). For this example, set the (x, y, z) position of this new GameObject as (-0.02,0,0), remember that the units are defined in meters so 0.02m represents 2cm. Then we have two FixPosDescriptors in the scene ("FixPosDescriptor" and "FixPosDescriptor2") pointing to different points in the virtual scene ([0.02,0,0] and [-0.02,0,0], see Figure 3.E and 3.F).

3. **Set the number of samples in the circular path**: select the CirclePositionDescriptor element on the Hierarchy and focus on the *CircleDescriptor.cs* inside the inspector tab, and set the "numSamples" field to 2000. These samples will make each particle move a bit faster than 2 full cycles per second each.

4. **Add *PrimitiveStateAsset.cs* component to both, Animation and Idle states:** Then, drag and drop the second primitive ("ParticlePrimitive2") on the "Primitive" field on both states. And finally, as both particles are using the same circle position descriptor to rule their motion, we need to define the index for each particle to start. Let's start with the "Animation" state, as we are drawing a circle of 0.04 m diameter (0.02 m radius), and we have the particles' initial location at 0.02 and -0.02 m on the X axis, we can then assume that they are on opposite sides of the circle, then let's define the first particle ("ParticlePrimitive") with the starting index 0 and the second particle ("ParticlePrimitive2") with the starting index in 1000. The case is different for The Idle state as the buffers attached to it are single-position buffers the index for FixedPosDescriptors should be always 0 (Figure 3.E).

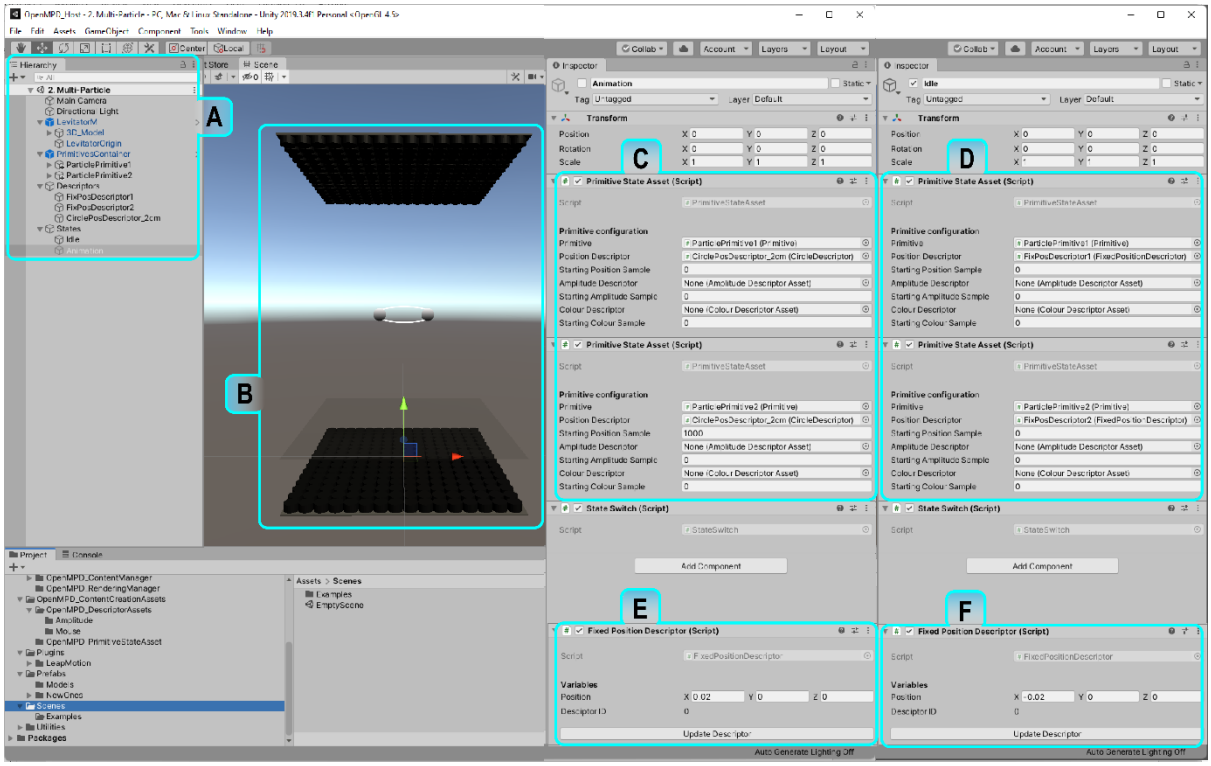

Figure 3: Example 3. Dual particle in a circular motion. A) shows the object structure in the virtual scene, B)shows the levitator 3D model and from C to F the components configuration is shown.

Same as we did in the previous example we need to click on the *InitiaState* and open the Inspector tab, as shown in Figure 3.C there is a *MPD_PrimitiveStateAsset* component attached to it, however, we now have two particles that we want to initialize together, for this example we need to add a second *MPD_PrimitiveStateAsset* component (one per primitive). To do so, we click on the "Add Component" button (see Figure 3.A, green box), at the bottom of the inspector tab and select the *MPD_PrimitiveStateAsset.* Then we can start dragging and dropping the first primitive and the first *FixPositionDescriptor to the first MPD_PrimitiveStateAsset* component on the Primitive and PositionDescriptor fields, same for the second primitive and second *fixPositionDescriptor*.

5. You are now ready to follow the last steps from Example 2 - Single Particle: Fast-moving Particles (PoV) (Steps 7, 8 & 9).

Note: to animate particles with independent behaviours it is best to keep particles separated at a distance >= 1cm to avoid merging the traps, because, once the traps are merged and the particles are together in a single trap, it is not possible to separate them again. This trap merging can be useful on animations; however, you need to be careful about how you use it.

Exercise E: *Replicating behaviours*. Now you can be able to replicate example 3 using independent circle position descriptors.

Exercise F: *Extending behaviours*. This is a good time to change and test the MPD capabilities, lets's try making each of the two particles draw a circle on different highs. Do you notice any issues when the particles are aligned vertically? Is there any effect when combining two fast-moving particles with direct manipulation?

4 EXAMPLE 4 - LEVITATED PROPS: MULTIPARTICLE STRUCTURES

This is Example 4 in this tutorial. In this example, we manipulate a levitated prop as a spatially constrained particle structure (a piece of fabric with some particles attached to it). This means that the spatial relationship between the particles in the arrangement is fixed (i.e., four particles in a square shape will maintain the shape when undergoing translations and rotations). Similar to **Exercise B** in Example 1 - Independent Particles: High-Level transformations. In other words, any transformations (translation, rotation and scale) applied to the parent will affect all the children as a group. Let's start with an empty scene and then please follow the steps listed below for this example. Please start following the Steps 1 & 2 from the Example 2 - Single Particle: Fast-moving Particles (PoV), then:

1. **Drag and drop into the de scene a MiniScreen prefab:** and set it as *Content* prefab child. Make sure the LeviProp is the centre of the levitator position (0,0,0), remember you can update it by clicking on the GameObject and on set the centre location on the "Transform" section inside the inspector. This particle has attached a *OpenMPD_Primitive* script to update particles' position and rotation to the physical levitator through the driver.
2. **Add a "MSFixDescriptor" prefab as child of the Descriptors node:** once you add it you need to define the initial position for each of the particles on the LeviProp. In this example we are using a 0.04m x 0.04m LeviProp size (4x4cm) then if its in the centre of the rendering space, we can assume that the particles on each corner will be at p1(-0.02,0.02,0), p2 (0.02,0.02,0), p3(-0.02,-0.02,0) and p4(0.02,-0.02,0). Then please expand the "MSFixDescriptor" and set each of the 4 "FixedPositionDescriptors" insides.
3. **Add an Empty GameObject as child of the "Descriptors" node**: Rename it as CirclePositionDescriptor and attach a *CircleDescriptor.cs* component to it, and similar to the Example 3, define as radius 0.02m and a number of samples 2000 samples per cycle.
4. **Add two empty GameObject as child of the States node**: Rename them as "LeviPropIdle" and "LeviPropAnimation", then attach four *PrimitiveStateAssets.cs and one StateSwitch.cs* script to each of them. As shown in Figure 4.C & 4.D, please assign the primitives inside the MiniSccreen prefab to each of the Primitives fields on both LeviPropIdle and LevipropAnimation states. Now let's set the starting position for each primitive on the LevipropAnimation state, similar to Example 3, we will set 0 to p1, 1000 to p2, 0 to p3 and 1000 to p4 (see Figure 4.C & 4.D). Leave amplitude and colour fields empty for now.
5. **Use a MPD device or simulator mode**: If you have a MPD device, this is the point where you need to make sure it is connected (USBs correctly recognized) and powered. If don't, then the unity visualization will reflect the indented particle behaviour when pressing play.
6. **Press play**: Once you press play, you would be able to visualize the LeviProp inside the virtual space in Unity. If the levitator is connected and correctly configured, you would be able to place the physical particles at each of the starting locations. You can use some twicers to position the physical particle in the correct location.
7. **Changing animation states**: To change between states you need to manually activate the Animation GameObject by activating the checkbox next to the GameObject name ("LeviPropAnimation" or "LeviPropIdle") in the inspector tab, and automatically the animation will switch from Idle to Animation state (and the Idle state will be then disabled) while the rendering of the animation will start.

By now the levitator (virtual and real) should be able to render four particles circular path with your defined radius and speed.

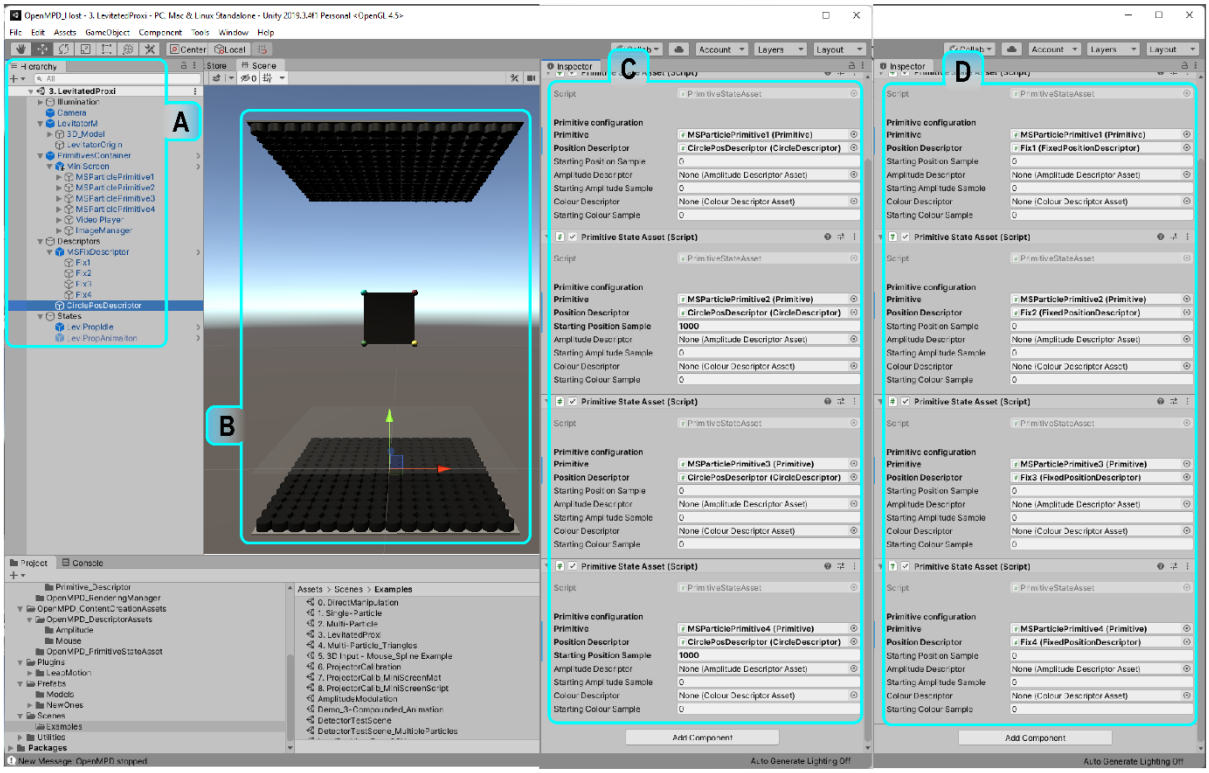

Figure 4: Example 4. LeviProp.

Exercice G: *Script Manipulation*. Now you can try adding a script and try controlling the position and rotation of the Leviprop by transforming the parent node ("MiniScreen") by using High-level updates (Unity transformations).

5 EXAMPLE 5 - MULTIPARTICLE TRANSITIONS

As shown in previous examples, low-level descriptors are an excellent way to improve particle displacements and reach higher particle speeds that would not be possible using high-level updates. However, OpenMPD support queued animations, where a correct transition between high-speed animations becomes crucial to not only maintain animation continuity but to avoid rendering failures (particle flying away from the target animation). Then an animation transition is an important element when working with queued animations in MPD systems, as they are the link between stages in the overall experience (considering stages as different animation/shapes/paths to go through). We can render multiple animations using a single particle by using the correct transitions to move across animations. In this example, we will use a set of three circles that will combine into one, but rotate and translate independently from each other. This is an example that takes advantage of our hybrid updates mode (low-level + high-level updates).472

Transition is an important element when working with animations MPD systems, as they are the link between stages in the overall experience, considering stages as different animation/shapes/paths to go through. We can render multiple animations using a single particle by using the correct transitions to move across animations. In this example, we will use a set of three circles that will combine into one, but rotating and translating independently from each other (see figure XX). As usual, we will start with an empty scene and then follow the Steps 1 to 6 from the Example 2 - Single Particle: Fast-moving Particles (PoV), then:

1. **Add another two primitives prefab as child of the *Contents* node**: set them all to (0,0,0) and rename them as " ParticlePrimitive1", "ParticlePrimitive2" and "ParticlePrimitive3".
2. **Add another two FixPosDescriptors prefab as a child of the Descriptors node**: Rename them as "InitialPos1", "InitialPos2" and "InitialPos3". For this example, set the x, y and z values for P1 as (0.01,0,0), P2 as (-0.01,0, 0.01732051) and for P3 as (-0.01,0, -0.01732051).
3. **Add three new empty gameObject as child of the *Descriptors* node**: Now change their names to "SubCircle1", "SubCircle2" and "SubCircle3". Then, attach a *circleDescriptor.cs* script to each of them as done in previous examples. In this example, we configured each of them with a radius = 0.01 and 2300 samples. The other circlePosDescriptor you added previously, configure it with a radius = 0.02 and 4500 samples.
4. **Duplicate the Animation state inside the States node**. Before configuring the three states available in the States node (Idle, Animation1 and Animation2), make sure they all have three *PrimitiveStateAsset.cs* components each (to hold n state per primitive in the scene). Now the first step is, assigning each particlePrimitive to a "primitive" field on the "Idle" state (use the ParticlePrimitive1, particlePrimitive2 and particlePrimitive3 in that order), the second step is assigning the corresponding fixPosDescriptors to each "positionDescriptor" field on the "Idle" state (make sure that particlePrimitive1 is related to fixPosDescriptor1 and so on). Next, we need to repeat step one for the animation states (Animation1 and Animation2), once we correctly assigned the particlePrimitives to their corresponding fields, we will proceed to assign the "CirclePosDescriptor" to all the three "positionDescriptor" fields inside the Animation1 state and for this example, let's use "startingPositionSample" as 0 for the first particle, 1500 for the second and 3000 for the third. And finally let's assign the subCircles ("SubCircle1", "SubCircle2" and "SubCircle3") to each of the "PositionDescriptor" fields on the "Animation2" state and let the "startingPositionSample" for all the elements in "Animation2" be 0 (see Figure 5.C-E).

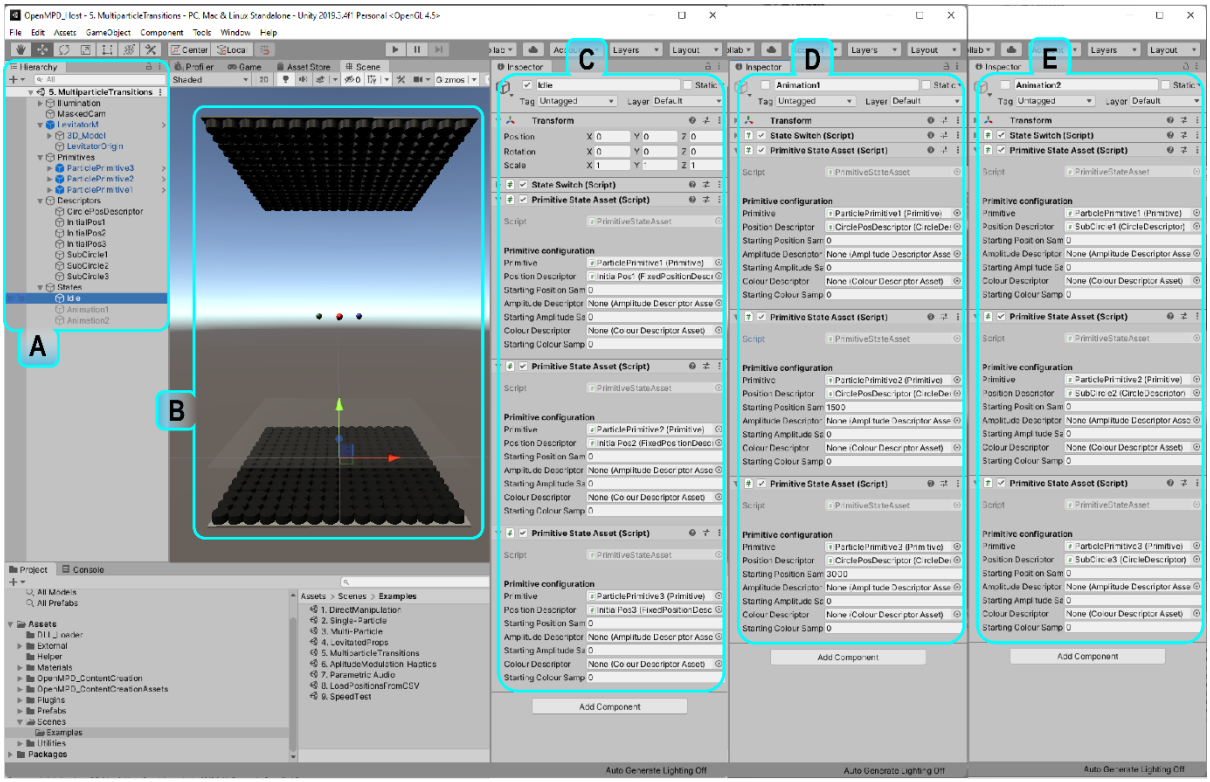

Figure 5. Example 5. Multiparticle transitions demo where “A” show the object's structure “B” the levitator 3D model with three particlesPrimitives. “C”, ”D” and “E” show the state's configuration.

5. **Press Play**. You can now activate the different animation states on the example, navigating across the states as you wish.

Important Note: OpenMPD supports descriptor’s Queuing, this means that the rule transition can be also done using this approach, using a descriptor to move a particle from one place to another and using a second fixed position descriptor to hold the particle in a different location to start a new animation. Figure 6 is an example of queued descriptors. In other words the “primitiveStateAssets” inside the state nodes are red as a queue of elements, then the required cyclic path for continuity will be required only for the last element in the queue.

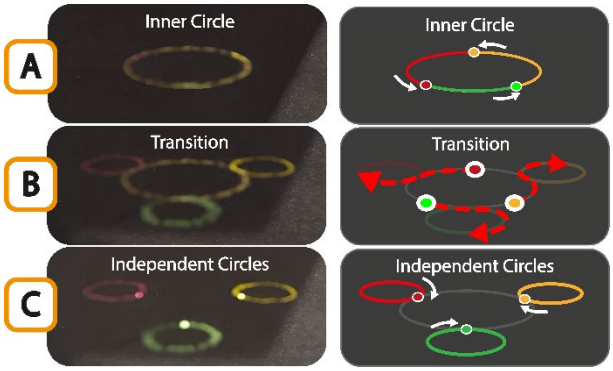

Figure 6: Dexterous manipulation via descriptor control: (A) Three Primitives rendering a PoV circle; (B) Primitives using accommodation paths to transition to a new initial position and adjust the speed; (C) Primitives in their new cyclic state, each showing a PoV circle.

Exercise H: *Hybrid*. After entering the play mode, and having the particles draw a circle, then click on the States node and the inspector you will see two scripts attached to it. Please tick the box “start sequence” inside the *CompoundedAnimationController.cs*. Now switch between animation states. What is the difference?

Exercise I: *Queue Transitions*. After this example, it's time now, star defining none-cyclic paths (paths where starting and final point are not the same), but remember you can use them to move across the animation, but always add a cyclic path right at the end of the queue to loop the last animation and avoid issues during the rendering state.

6 EXAMPLE 6 - AMPLITUDE MODULATION: HAPTICS

In this example we show how haptics can be generated using OpenMPD, by literature haptics can be done mainly by using one of two methods,

i) Spatial Modulation (SM): Modulating the positioning of the particles. I mean, making the trap cross the same point X number of times per second, where X is the modulation frequency. It is important to know that haptic sensations on the human hand are usually rendered using a modulation frequency of ~200Hz, due to the skin's ability to perceive such frequency (mechanoreceptors on the skin). Variations in the modulation frequency will vary how we perceive the sensation, and these changes can be used to render textures in id-air.

ii) Amplitude Modulation (AM): Modulating the acoustic trap amplitude. As we have seen in the previous examples, OpenMPD supports Position, colour and amplitude descriptors. By modulating amplitude we can use a static trap position and vary the trap's amplitude over time (using a sinusoidal behaviour), ensuring having as many full cycles as the modulation frequency (~200Hz).

AS spatial modulation can be done by using our Example 2 - Single Particle: Fast-moving Particles (PoV) and adjusting the spatial modulation frequency. Then, in this example, we focused on option ii, "Amplitude Modulation". To do it, we will use an amplitude descriptor to generate a haptic/tactile point where we will be able to control the frequency to generate different stimulation types on the hand's mechanoreceptors.

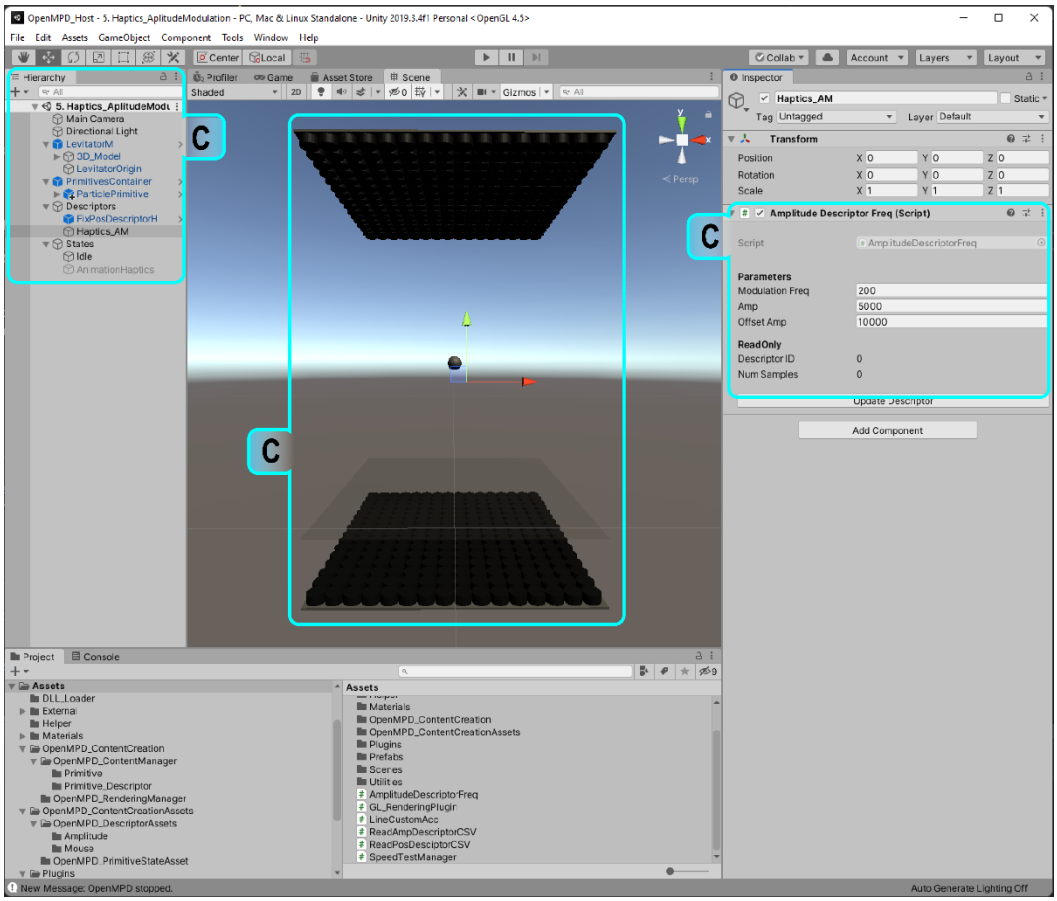

Figure 7: Example 6. Amplitude modulation virtual scene. A) shows the structure of the virtual object in the scene, B) shows the levitator with the primitive in the centre of the rendering space and C shows the *AmplitdeDescriptorFreq* module and the variables to configure the amplitude modulation for this example.

Important Note: OpenMPD supports the "phase-only" mode, which saves computation storage and performance o when amplitude does not need to be computed. Then make sure you deactivate this option when working with amplitude descriptors (LevitatorOrigin> OpenMPD_PresentationManager>PhaseOnly).

For this example we will open a new Scene in Unity and follow Example 2 - Single Particle: Fast-moving Particles (PoV), steps 1 to 6. Now, our scene is almost done, however, we are missing an important part of this example and that is the declaration of the "AmplitudeDescriptor". Then:

1. **Rename the "CirclePosDescriptor" GameObject to "HapticsAM"**: and remove the *CirclePosDescriptor.cs* by right-clicking on the three dots on the right corner of the script section inside the Inspector tab, and selecting "Remove component". Then attach an *AmplitudeDescriptorFreq.cs* module instead. This particular descriptor will allow you to edit three parameters; modulation frequency ("modulationFreq"), amplitude to use ("amp") and the amplitude offset in the output signal ("offsetAmp"). OpenMPD does not set a maximum amplitude value, as this value would vary with the hardware used, a common range used to set this variable is between 15000 Pa and 20000 Pa. In this example we used a "modulationFreq" = 200Hz, an "amp" value of 5000 (this means +- 2500 Pa from the offset value), and an "offsetAmp" of 10000Pa (see Figure 7.C).

CodeExample 3

```
public class AmplitudeDescriptorFreq : AmplitudeDescriptorAsset
{
    [Header("Parameters")]
    //public Vector3 vector3 = new Vector3();
    public float modulationFreq = 200;
    public int amp = 5000;
    public float offsetAmp = 10000;

    [Header("ReadOnly")]
    [ShowOnly] public uint descriptorID = 0;
    [ShowOnly] public int numSamples = 0;
    [HideInInspector] public float[] amplitudes;

    [ButtonMethod]
    private string UpdateDescriptor(){
        updateDescriptor = true;
        return "Amplitude Descriptor: Update requested";
    }

    // local variables
    bool updateDescriptor = false;

    // Start is called before the first frame update
    void Start() {
        updateDescriptor = true;
    }

    // Update is called once per frame
    void Update() {
        if (OpenMPD_PresentationManager.Instance() && updateDescriptor) {
            GenerateAmplitudes();
            updateDescriptor = false;
        }
    }

    void GenerateAmplitudes() {
        numSamples = (int)(10000 / modulationFreq);
        float[] amplitudes = new float[numSamples];
        for (int s = 0; s < numSamples; s++)
            amplitudes[s]=offsetAmp+(float)(amp*Math.Cos((2*Math.PI*s)/numSamples));

        descriptor = new Amplitudes_Descriptor(amplitudes);
        descriptorID = descriptor.amplitudesDescriptorID;
    }
}
```

CodeExample 3 shows the *AmplitudeDescriptorFreq.cs* script, which keeps the same structure mentioned in previous CodeExamples. As you can see, it is quite similar to the *CircleDrescriptor.cs* component described in the *CodeExample 2*, however, here, the "AmplitudeDescriptorFreq" class is inheriting from *AmplitudeDescriptorAsset.cs* instead of *PositionDescriptorAsset.cs,* and unlike the position descriptor where each position was stored using a set four floats (homogeneous coordinates), Amplitude descriptors define the amplitude value using a single float per element (amplitude per position). In this script, the method in charge of computing the AM is "GenerateAmplitudes()" in which, the input values (modulation frequency, max amplitude and amplitude offset) are used to compute a sinewave-like output used in this example.

2. **If we have an MPD device connected we can now press play:** once you enable the "AnimationHaptics" state, the haptic sensation should be rendered on the location of the acoustic trap, so make sure it is in the centre of the rendering space.

Exercise J: *Modulation Settings*. The best way to see how different modulation frequencies can be felt by the human hand is by trying it yourself. You can start trying frequencies in the range of [50 - 300]. What do you expect to feel when increasing or decreasing the frequency? You can now try changing the amplitude values in the range of [100 - 10000].

Exercise K: *Motion + Haptics*. The next thing to try is using a circle descriptor + amplitude descriptor. I mean, use a circle descriptor with a motion circle (slow rotations will be better to make sure you feel where the stimulus is).

Exercise L: *Hybrid mode*. Finally, try using two primitives, one to levitate a particle and the second one to generate haptics. How is your device performing? How close can you have them without destructive interference? How fast can you move the levitated particle when having the haptic stimulus active?

7 EXAMPLE 7 - PARAMETRIC AUDIO GENERATION

This example will highlight the OpenMPD capability of supporting parametric audio applications. While the standard audio sampling rate is ~44kHz, OpenMPD works at 10kHzas it is the update rate set as standard (this depends on the hardware). Then the audio file we will play in this example will be resampled to 10kHz (if your MPD device operates at a lower update rate the audio file should be resampled to it). Similar to Example 6 - Amplitude Modulation: Haptics OpenMPD uses the amplitude descriptor to play the audio file. This means that if you have the amplitude array of your audio file you can follow the Amplitude modulation example and instead of using *AmplitudeDescritporFreq.cs* to generate the array of amplitudes, use a *ReadAmpDescriptorCSV.cs* to read your file and use it as any amplitude descriptor. It is not recommended to levitate and play parametric audio on the same acoustic trap as the modulation may cause instabilities in the particle. It is important to mention that the processed audio files (.csv) must be stored inside the "PAudio" folder (in Unity's root).

As the procedure to play parametric audio files is quite similar to the one described in Example 6 - Amplitude Modulation: Haptics, we will explain our method to process the audio file to generate the AM array out of it. However, the tool we need is not contained in the Unity implementation, but in the low-level version of it. Then we will need to open the "OpenMPD_Native" solution which is the C++ version for the framework, we use Visual Studio 2019 for this implementation, while other VS versions can be compatible, using this the vS2019 version is suggested. Then follow the steps listed below:

1. **Open the "OpenMPD_Native" solution**: double click on the *OpenMPD.sln*, once opened, go to the "SpeedTests_OpenMPD" project and set it as the current project, this can be done by right-clicking on it and selecting "Set as start-up project".
2. **Make sure the "AmplitudeModulationMain" is not omitted from the build:** this can easily be visualized by looking to the left of the file name inside the solution explorer, if there is a red icon, this means that it is omitted from the compilation (see Figure 8)
3. **Configure the MPD device boards' IDs:** same as configuring the setting on the LevitgatorOrigine node, here you will find right under the headers, a section with the main configuration variables (FPS_Devider, geometries, tobBoard, bottomBoard, ForceSync, etc.). For now, leave the configuration as it is and just update IDs of your device.
4. **Resample and modulate (AM) the audio file:** As explained before the standard audio sampling rate is ~44kHz, and the sampling rate should match the update rate of your device, then in our case OpenMPD uses 10kHz update rate, and that is the sampling rate we will use. Using an audio sampling rate different from the update rate from your device will end up in distorted audio signals. Then just input the audio file name on the "fileName" variable. The file should be a .wav file and it must be stored in the LIBS_HOME/bin/x64 folder. After pressing play, OpenMPD will read and process your audio file, and it will write a comma separated file as a result of the process. We use double sideband modulation for AM, however other approaches can also be used (e.g., single sideband modulation).

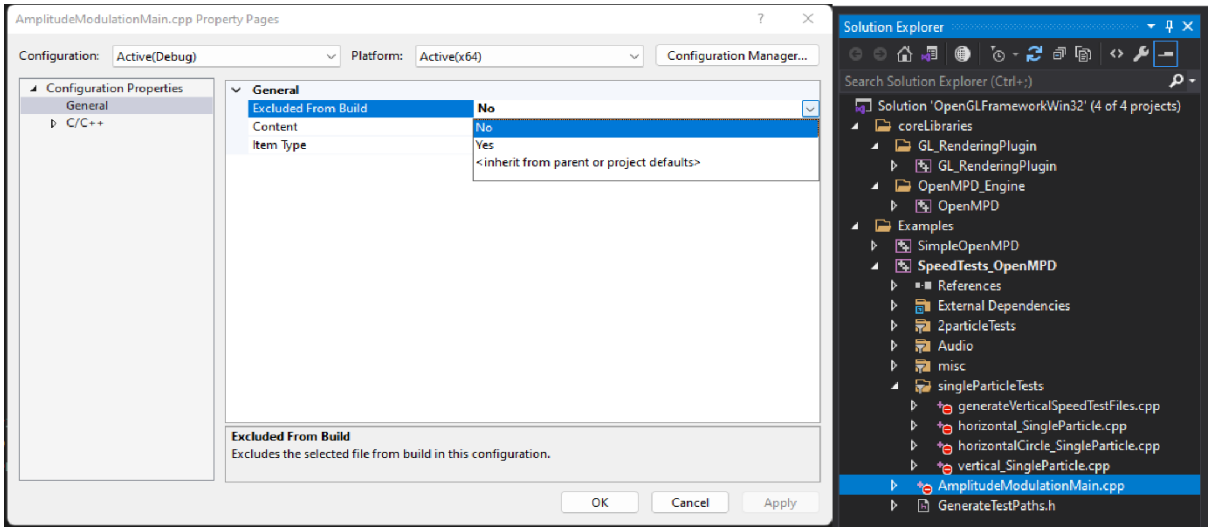

Figure 8: Example 8. The solution explorer is shown on the right with the AmplitudModulationMain file selected inside the SpeedTestsOpenMPD. On the left, a window to configure the exclusion from the build is shown.

Exercise M: *Audio testing*. Try playing the different files stored in the PAudio folder. And use the steps described in this example to generate your own.

Exercise N: *Acoustic interference*. Try to have a levitated particle + a parametric sound trap. What do you observe when playing? What happens if you add a haptic trap as well?

8 EXAMPLE 8 - READING DESCRIPTORS FROM A CSV FILE

As we have observed, OpenMPD takes arrays of data to describe the primitives' behaviour in form of descriptors. So far the descriptors were generated inside the framework, however, OpenMPD supports a descriptor definition from external files. In this example, we will show the implementation to generate position descriptors from CSV file data. OpenMPD also has the same functionality for colour and amplitude descriptors.

For this example, we will open a new Scene in Unity and follow Example 2 - Single Particle: Fast-moving Particles (PoV). Once the steps 1 to 6 are complete then:

1. **Rename the "CirclePosDescriptor" GameObject to "PosDescFromCSV":** now remove the *CircleDescriptor.cs* component and attach an *ReadPosDesciptorCSV.cs* to it (See Figure 9.A). This descriptor only requires the name of the file from where the array of positions will be extracted. This descriptor will automatically search for the target file once OpenMPD is instanced. You can also load different paths from once in play mode by changing the file name and pressing the "updateDescriptor" button.

2. **Duplicate the FixpositionDescriptor and rename it as "FixpositionDescriptorAdj":** please remove the *FixedPosDescriptor.cs* and attach a *UpdateFixPosDescriptors.cs* (See Figure 9.A). The only difference between these two components is that the second read the fixed position from a reference object, for instance, it read the first position from a circle position descriptors and keeps it as its fixed position. Now drag and drop the "PosDescFromCSV" GameObject into the "refDescriptor" field.

Important Note: This script is looking for the files inside the "OprimizedPaths" folder stored in Unity's root folder (OpenMPD_framework\OpenMPD_Host\OprimizedPaths). Then, any external CSV file must be stored in the same folder for this example to read it. You can also code your own read from file script, as this is just an example of how to do it, however many other ways can be explored.

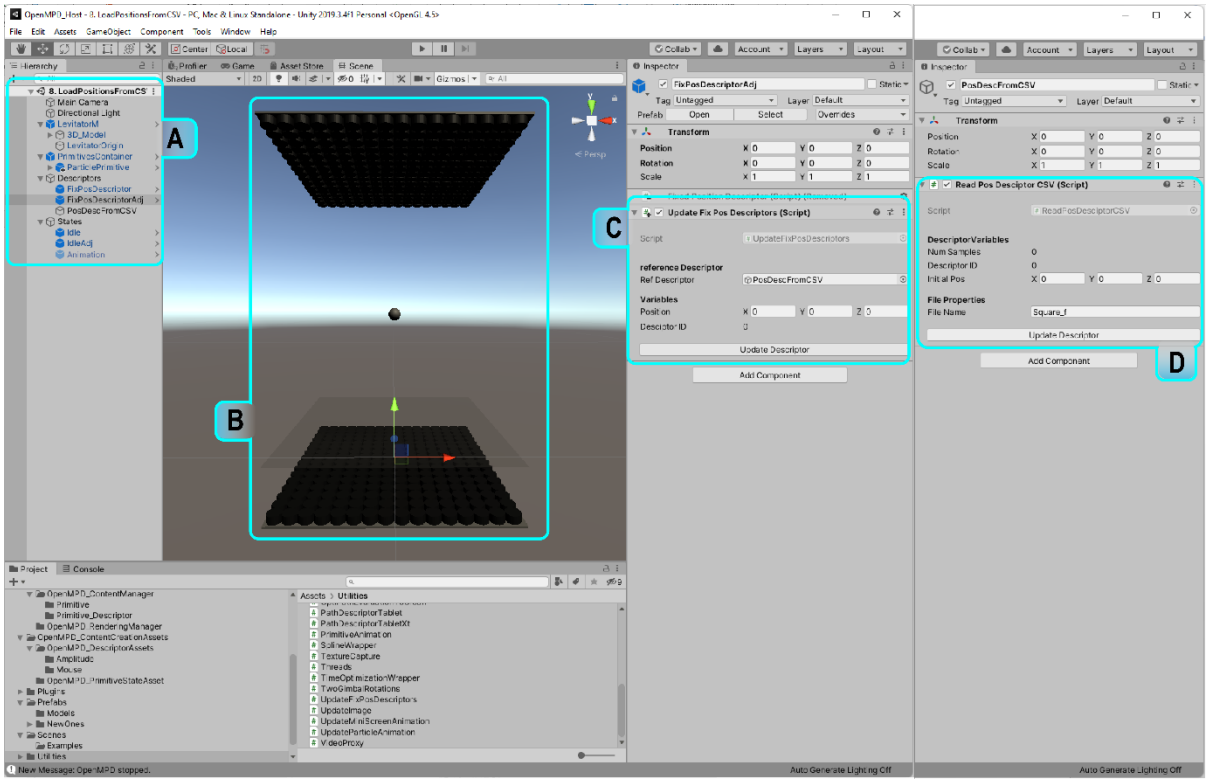

Figure 9: Example 8. Load position descriptors from a file where A) shows the object structure in the virtual scene, B) shows the virtual levitator with a single particle in the centre of the rendering space, C) shows the UpdateFixedPosDescriptor to automatically update a fixed position from other descriptors set as reference and D) shows the *readPosDescriptorCSV.cs* component to read the date and declare a descriptor from an external CSV file.

3. **Configure the Animation state**: Drag and drop the “PosDescFromCSV” to the positionDescriptor field.
4. **Now you can press play**: once enter in play mode the positions from the file are read and the position descriptor generated, however, as the fixed position descriptors were probably generated before the reading stage from the CSV file was finalized, the fixed position descriptor should be updated with the new value from the new positions array. This means once you press play, just select the “FixPosDescriptor” GameObject from the Hierarchy tab to enable the edition of the properties in the inspector tab, then press the “updateDescriptor” button on the inspector tab (See Figure 9.). And finally enable the “IdleAdj” state, after this step you can now place the particle on the starting position from the *UpdateFixPosDescriptors.cs.* once this was done you can start now switch between “Animation” and “IdleAdj” to play and stop your animations.

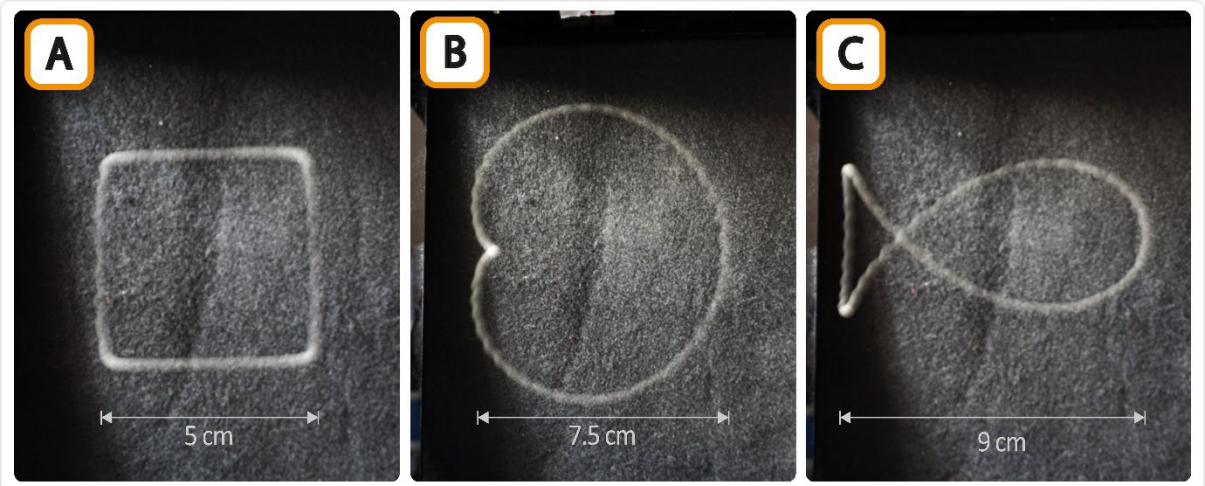

Figure 10: Example 8. A, B and C show the Physical rendered paths using the ReadFrom CSV (square, cardioid and fish).

The shapes in Figure 10 are optimized paths used as examples, these files are stored inside the “OptimizedPaths” in the Unity root folder. This is the last example in this tutorial, and we will be updating OpenMPD documentation and functionality, make sure you have the latest version of both.