Csound and Unity3D

Rory Walsh
Ireland

Adaptive audio for games presents a serious challenge for both developers and sound designers. Existing audio middleware offer some solutions, but fall way short of the kind of tools sound designers are used to working with. This text will focus on the use of Csound as a fully integrated sound engine for the Unity(3D) game engine.

It should be noted that this text is written from the perspective of a musician and sound designer rather than a game developer. Nonetheless, it does try to cover as much of the core principles of game design in Unity as is possible within the restrictions of a single text. As the main Csound wrapper for Unity is written in C#, this is the language that all scripting examples will be shown in. Readers should have some basic understanding of Csound and C#.

1 Core Concepts

Game engines are extraordinary development environments. The relatively ease in which they combine low level programming with high level graphics abstraction is nothing beyond impressive; Unity3D in particular. Due to the complexity of the software environment, it's important that users get to grips with the fundamentals as quickly as possible.

1.1 Assets

Most projects in Unity can be broken into the following Assets.

1.1.1 Scenes

Scenes can be thought of as game levels, and every Unity project must have at least one scene. Most of the simple examples in this text have only one scene in them. By default Unity will create a basic scene each time you create a new project. It's best to save the scene at the very start of a project. A scene will contain a 3D world. Be default this world will be empty, but game objects can be dragged into the scene at any point during development.

1.1.2 Game objects and prefabs

The GameObject class is the most basic building block in Unity. Games may be comprised of 1000's of different GameObjects. While games are made up of GameObjects, GameObjects themselves are made up different *Components*. For instance, all GameObjects have a transform component that determines where in the 3D space they are placed. Some will have physics components attached while others may have animation components.

The most basic types of 3D GameObjects are *cubes, spheres, capsules, cylinders, planes and quads.* Any number of these objects can be grouped together to create prefabs, which can be instantiated and used across any number of scenes. They can also be exported and used in another other Unity games one may be developing.

1.1.3 Scripts

The aforementioned GameObject behavior can be controlled using special *Components*. While the variety of *Components* provide large amount of control via the Properties Panel within Unity, real control is provided through *Scripts*. *Scripts* are files that contain instructions on how GameObjects should behave and interact with the environment around them. They are key to developing games in Unity. Unity scripts can be written in C# and Unity's own UnityScript language which is heavily based on Javascript. The examples presented in this text are written in C#. It is not necessary to learn how to program in C# or Javascript before learning Unity. It is entirely possible to learn C# or Javascript while scripting in Unity.

1.1.4 Materials and shaders

Materials determine how the surface of a GameObject will be rendered, and are dependent on shaders. Shaders are scripts that determine how all the GameObject's pixels will be coloured.

1.1.5 Project Hierarchy

It is generally a good idea to create a clear project hierarchy whenever starting a new project in Unity. To do so, simply add folders titled 'Scenes', 'Prefabs', 'Scripts' and 'Materials' to Unity's *Project* tab.

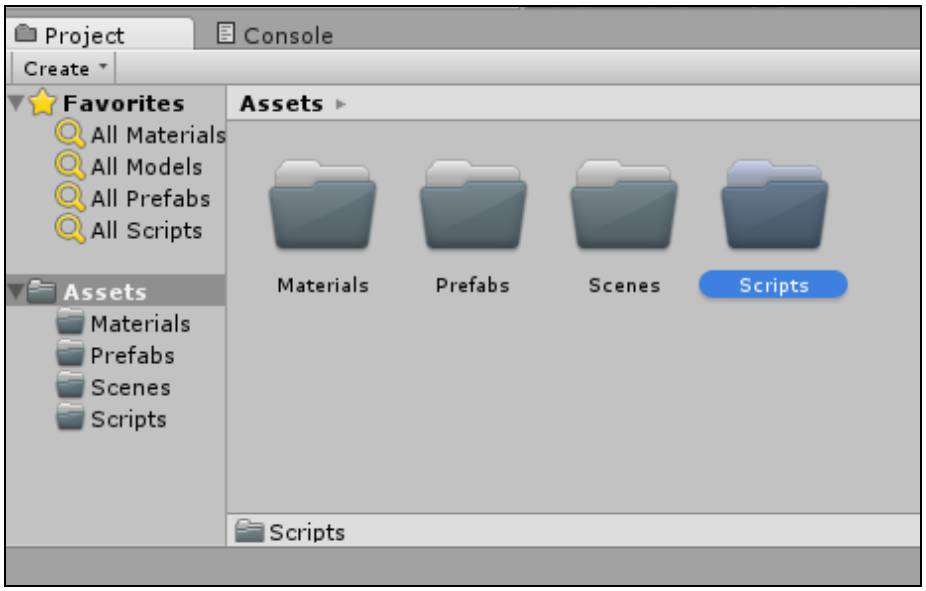

Figure 1 Project hierarchy

1.2 GameObject

GameObjects are used to construct things that appear on screen. All objects you see in a game are constructed using GameObjects. Unity provides several different types of GameObjects such as cubes, spheres, capsules, cylinders, planes and quads. While GameObjects themselves are quite primitive, several can be combined to create more complex shapes and models. Each GameObject has several *Components* that determine how it will appear and behave on screen. Each *Component* will add a further level of control over the object. Mesh renderer components for example provide ways of controlling how an object will be rendered. Physics components provide control over things such as gravitational force and drag. Scripts can also act components, providing further control over an object.

Every GameObject will have a *Transform* component. It is a fundamental component that will control the position, scale and rotation of the GameObject.

Property	**Function**
Position	Position of the Transform in X, Y, and Z coordinates.
Rotation	Rotation of the Transform around the X, Y, and Z axes, measured in degrees.
Scale	Scale of the Transform along X, Y, and Z axes. Value “1” is the original size (size at which the object was imported).

Each GameObject will also have a *Tag* property. *Tags* are labels that are used to identify different GameObjects, especially when scripting GameObject behaviours.
It is important to understand the GameObject-Component paradigm in Unity. Without associated components, GameObject would be restricted to static objects with no means of interacting with the environment around them. The more components added to a GameObject, the more that GameObject will be able to do within the game.

1.2.1 Coordinates

Two common coordinate systems that appear quite often in Unity are *Local* and *World* space. The system being used depends on the position of the origin. When moving an object around in world space, its XYZ coordinates are absolute values. When moving an object in local spaces, all coordinates are relative to the object's point of origin. Consider an empty room with a single chair in it. Every time the room rotates; the chair within it will rotate too. In this case the chair is being rotated in *World* space. However, the chair can also be rotated on its own using *Local* space. In this case the chair will rotate around its own point of origin.

The position property of the aforementioned *Transform* component controls the GameObject’s position in world space, while *Rotate* and *Scale* are set to local space by default.

1.3 Simple Movement

Being able to move a player character around the screen is something that is needed for the vast majority of games, whether in 2D or in 3D. The easiest way to move an object

around the screen is to script the movements manually. To do so, a new component needs to be added to the GameObject that will represent our player. In this case, the player will be represented by a simple sphere.

The sphere should behave the same way in the game world as it would in the real world. It will need to roll around the place according to external forces such as gravity. To simulate real-world physics, a component known as a *RigidBody* will need to be added to the sphere. With the RigidBody component in place, the sphere now has an associated mass and gravitational field attached to it.

A simple script can be added to the GameObject sphere to control its movement. Scripts can be added to GameObjects in the very same way as any other components. When a new script is created, Unity will automatically generate some template code in the form of an empty class that is derived from the MonoBehaviour. Every script in Unity must be derived from the MonoBehaviour base class. The following is a simple player movement script written in C#.

```
using UnityEngine;
using System.Collections;

public class PlayerController : MonoBehaviour {

    public float speed=50;
    private Rigidbody rBody;

    void Start()
    {
        rBody = GetComponent<Rigidbody>();
    }

    // Update is called once per frame
    void FixedUpdate ()
    {
        float moveHorizontal = Input.GetAxis ("Horizontal");
        float moveVertical = Input.GetAxis ("Vertical");
        Vector3 movement = new Vector3 (moveHorizontal, 0.0f, moveVertical);
        rBody.AddForce(movement * speed);
    }
}
```

The script above declares two class member variables. One is **speed**, which controls the speed of the sphere's movement. This member variable is declared as public. This means that it can be accessed and updated from the script component's property tab in the main Unity interface. This saves one the hassle of having to open, change, and recompile the script each time a member variable needs to be updated.

The second class member variable is of type RigidBody. In the **Start()** menu, **rBody** is assigned the RigidBody component that is attached to the sphere. The start method is called on the first frame that the script is active. It is called before any of the update methods, and is only called once.

The **FixedUpate** function is called once per frame. The static function **Input.GetAxis()** can be used to retrieve the value of the virtual axises. **GetAxis()** will return a value in the range of -1 to 1. These values are then used to instantiate a new Vector3 object which is then passed to the **AddFoce()** method of the sphere's RigidBody component.

1.3.1 CharacterController

In certain cases, especially when working with first person games, a Rigidbody component can prove a little overwhelming. Without a strong understanding of Unity's Physics components, your main player object is likely to start bouncing and spinning out of control. In most cases it is easier to avoid the RigidBody approach and use a *CharacterController* component instead.

A GameObject with a CharacterController is far easier to control. Instead of adding forces to move a player character, a CharacterController can be instructed to simply move around the scene. In order to use a character controller, simply add one to a GameObject but make sure that the GameObject does not have a Rigidbody attached to it. CharacterController provides a very simple method for moving a GameObject around the scene. It is called **SimpleMove()** and is shown below in a sample script from the Unity reference manual.

```
using UnityEngine;
using System.Collections;

public class PlayerController : MonoBehaviour
{
    public float speed = 3.0F;
    public float rotateSpeed = 3.0F;

    void Update()
    {
        CharacterController controller = GetComponent<CharacterController>();
        transform.Rotate(0, Input.GetAxis("Horizontal") * rotateSpeed, 0);
        Vector3 forward = transform.TransformDirection(Vector3.forward);
        float curSpeed = speed * Input.GetAxis("Vertical");
        controller.SimpleMove(forward * curSpeed);
    }
}
```

The **GetComponent()** method is used to access the CharacterController component that is attached to the GameObject. **transform.Rotate()** is passed the value returned by **GetAxis()**(see above). Remember that all GameObjects in Unity have a transform component attached. Each GameObject's transform component can be accessed directly without have to use the **GetComponent()** method. A **Vector3** named forward is then assigned the vector returned from **transform.TransformDirection()**. The**TransformDirection()** method is needed to convert the character's direction, in local space, to world space. To understand the difference between local and world direction, try replacing the last line of the**Update()** method with this:

```
    controller.SimpleMove(Vector3.forward * curSpeed);
```

When in game mode, the player's forward direction is no longer changing as the player rotates. This is because SimpleMove() takes a world space directional vector, not a local one.

1.4 Cameras

Cameras define what is shown on screen. Every scene must have at least one camera. Unity takes care of this fact by populating each scene with a default camera object.

Cameras can be moved and manipulated just like any other GameObject. They can also be attached to other GameObjects. Each camera has a view frustum which determines what will appear in the camera's field of view. View frustums have a *near*, and *far* field that set the nearest and furthest points that the camera can display. Any objects that are placed outside the viewing frustum with not be rendered.

Unity provides two camera modes:

- **Perspective** In perspective mode, cameras will behave just like a real world camera. The further away from the camera the object is, the smaller it will appear.
- **Orthographic** In orthographic mode there is no sense of perspective. The viewing frustum is a cuboid rectangle and all objects are rendered in parallel. This mode can be used for 2D games and isometric games such as board games, puzzle, and strategy based games.

The camera's parameters can be adjusted using the camera's property editor. Both *Near* and *Far* clipping planes can be controlled, while the *Field of View* parameter controls how wide the camera angle will be.

1.4.1 Moving the camera

Cameras can be moved using simple scripts that can be attached to them as components. In this simple script, the camera will follow a public GameObject. Any classes or variables declared as pubic can be accessed within Unity's main editor.

```
using UnityEngine;
using System.Collections;

public class CameraController : MonoBehaviour {

    public GameObject ball;
    // Use this for initialization
    void Start ()
    {

    }

    // Update is called once per frame
    void Update ()
    {
    transform.position      =      new      Vector3(ball.transform.position.x,
ball.transform.position.y+3, ball.transform.position.z - 10);
    }
}
```

1.4.3 First Person Cameras

Unity ships with some simple utility scripts that can be used to control different aspects of how cameras behave. One such script is the *SmoothFollow* script which can be imported through the *Scripts* package. Once the script has been imported, it can be dragged to the scene's camera. The *SmoothFollow* script's*Target* property should be assigned the GameObject that the camera will follow. In most cases, this GameObject will be the player character. Note that this script works best when a CharacterController is attached to the player. *SmoothFollow* also provides properties to control the camera's distance and height from the player.

1.5 Collisions and Triggers

Collision detection is a major aspect of game mechanics. Almost every action in a game involves some level of collision detection. Without collision detection the game's main characters would be able to walk through walls while enemies could roam at large in the knowledge that they can never be killed! Unity provides several different collision detection mechanisms.

Each time we populate a scene with a new GameObject, Unity will automatically attach a *Collider* component to the object. By adding a new method to the previous code we can test for collisions, but first one must decide what type of collision to detect.

1.5.1 Triggers

When 'Is Trigger' is ticked in the *Collider* property panel, collisions are set up as triggers. This means that when two objects collide, they will pass through each other rather than rebound as physical objects would. These types of collisions are typically used for collecting items rather than bouncing off them. The **OnTriggerEnter()** method can be used to detect collisions.

```
using UnityEngine;
using System.Collections;

public class PlayerController : MonoBehaviour {

    (...)

    void OnTriggerEnter(Collider other)
    {
        Debug.Log("A trigger event has taken place between this GameObject and
"+other.name);
    }

    (...)
}
```

1.6.2 Triggers

When the 'Is Trigger' attribute of a Collider component is disabled, GameObject's will collide with each other just as real-world objects do. The **OnCollisionEnter()** method is used to detect if a collision has taken place.

```
using UnityEngine;
using System.Collections;

public class PlayerController : MonoBehaviour
{

    (...)

    void OnCollisionEnter(Collision other)
    {
        Debug.Log("A collision has taken place this GameObject and
"+other.gameObject.name);
    }

    (...)
}
```

1.5.2 CharacterController collisions

The previous section on *Movement* introduced the CharacterController component. A GameObject with a CharacterController attached can use a special method called **OnControllerColliderHit()** to detect collisions between it and other objects. The main advantage of this particular method is that it is called whether the colliding object has a Rigidbody component attached or not. Therefore it will report all collisions that have take place between CharacterControllers and all other GameObjects in a scene.

```
using UnityEngine;
using System.Collections;

public class PlayerController : MonoBehaviour
{

    (...)

    void OnControllerColliderHit(ControllerColliderHit hit)
    {
         //Debug.Log(hit.gameObject.name);
    }

}
```

2 CsoundUnity

CsoundUnity is a custom Csound package for Unity. It provides Unity users with a means of accessing Csound's core API within their Unity C# scripts. Borrowing from Richard Henninger's Csound6 .Net framework, CsoundUnity provides a Unity bridge to a set of pInvoke signatures that are used to call Csound's native API functions from C# code. The system works on OSX and Windows. The following sections will cover the steps involved in setting up and using CsoundUnity for the first time.

2.1.1 Setting up

CsoundUnity packages for both OSX and Windows can be downloaded from https://github.com/rorywalsh/CsoundUnity/releases. Once the package has been downloaded, it needs to be imported directly into Unity by accessing the 'Assets' menu item and scrolling down to 'Import Package'. Select 'Import Custom Package' and browse to the download location of CsoundUnity.

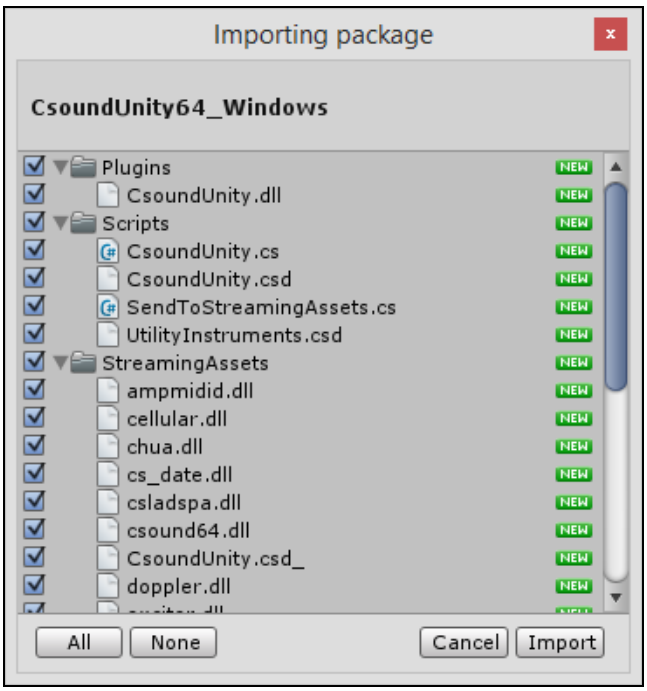

Figure 2 Importing CsoundUnity Package

During import, CsoundUnity will modify the project hierarchy by creating folders titled *Scripts*, *Plugins* and *StreamingAssets*, if they do not already exist. The following files/folders will be imported/created:

- **Assets/Plugins/CsoundUnity.dll**

This is the Unity C# .DLL that is responsible for loading Csound. The CsoundUnityBrdige class defined in this library is accessible within Unity scripts.

- **Assets/StreamingAssets**

Csound's core dlls will get installed to this location. On Windows this will be a list of Csound specific .dlls. On OSX this will be the Csound framework. This folder gets exported whenever a game is exported as standalone.

- **Assets/Scripts/CsoundUnity.cs**

This is the main Csound script in CsoundUnity. It creates an instance of CsoundUnityBridge and provides access to a handful of Csound's most useful methods. CsoundUnity does not attempt to expose all of Csound's native API functions. For the sake of simplicity it focuses just on those methods that are most useful such as compiling, message input/output, and channel communication.

- **Assets/Scripts/CsoundUnity.csd**

This file provides some simple instrument definitions that are used by various methods in the CsoundUnity class. This file should be included in the header section of the project's main Csound .csd file by using the following #include statement.

```
#include "CsoundUnity.csd_"
```

- **Assets/Scripts/SendToStreamingAssets.cs**

All Csound files should be placed in the *Scripts* folder. This simple scripts simply copies all Csound files to the StreamingAssets so they can be read by Csound.

2.2 The CsoundUnity class

The CsoundUnity.cs script provides access to the CsoundUnityBridge library, which in turn exposes various native methods of the Csound API. This script should be instantiated when a game first starts. The easiest way to ensure this happens is by attaching it to game's main Camera component. Most users of CsoundUnity will not need to edit this script at all, but more advanced users may wish to add their own utility methods to it.
The CsoundUnity class derives from MonoBehaviour. All Unity classes must derive from MonoBehaviour. CsoundUnity contains only 3 public member variables.

```
using UnityEngine;
using System.Collections;
using System.Runtime.InteropServices;

(...)

public class CsoundUnity : MonoBehaviour {

    // Use this for initialization
    private CsoundUnityBridge csound;
    public string csoundFile;
    public bool logCsoundOutput=false;

(...)
```

The private member variable **csound** provides access to the CsoundUnityBridge class, which is defined in the *CsoundUnity.dll*(Assets/Plugins). If for some reason CsoundUnity.dll cannot be found, Unity will report the issue in its output Console. The CsoundUnityBrdige object provides access to Csound's low level native functions. The **csound** object is defined as private, meaning other scripts cannot access it. If other scripts need to call any of Csound's native functions, then methods should be added to the CsoundUnity.cs file.

The public string variable **csoundFile** should be given the name of the project's .csd file. CsoundUnity only loads single .csd file, however, this .csd file can contain numerous includes for other .csd files. **logCsoundOutput** is a Boolean variable. As a Boolean it can be either true or false. When it is set to true, all Csound output messages will be sent to the Unity output console. Note that this can slow down performance if there is a lot of information being printed.

As the variables **logCsoundOutput** and **csoundFile** are public, they can be accessed within through the CsoundUnity component panel as shown in the following screenshot.

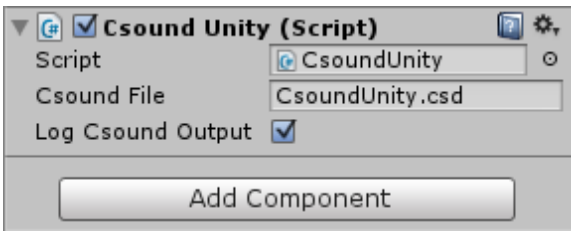

Figure 3 CsoundUnity component panel

Unity classes avoid using constructors to initialise and instantiate data. Instead they use an Awake() methods. The Awake() method for CsoundUnity is shown below.

```
void Start ()
    {
        string csoundFile =
Application.streamingAssetsPath+"/"+csoundFile+"_";
        string dataPath = Application.streamingAssetsPath;
        System.Environment.SetEnvironmentVariable("Path",
Application.streamingAssetsPath);
        csound = new CsoundUnityBridge(dataPath, csoundFile);
        csound.startPerformance();
        csound.setStringChannel("AudioPath",
Application.dataPath+"/Assets/Audio/");

        if(logCsoundOutput)
            InvokeRepeating("logCsoundMessages", 0, .5f);

    }
```

The CsoundUnityBridge object is responsible for creating and compiling an instance of Csound. It takes a path to the project's Data folder, and the .csd file that Csound will need to compile. **startPerformance**()will start a performance thread to run Csound, while **setStringChannel**() will pass the full path of the project's *Audio* directory to Csound on a channel named 'AudioPath'. This is useful when working with audio samples.

If **logCsoundOutput** is set to true, CsoundUnity will print any new Csound output messages to Unity's output Console using the following method.

```
void logCsoundMessages()
{
    //print Csound message to Unity console....
    for(int i=0;i < csound.getCsoundMessageCount();i++)
        Debug.Log(csound.getCsoundMessage());
}
```

CsoundUnity also provides a host of useful methods for interacting managing and interacting with sound files. Some of these are shown in the next section, but more details can be found at http://rorywalsh.github.io/CsoundUnity/

2.3 Triggering instruments and playing sounds

As mentioned in the previous section, CsoundUnity needs to be imported before it can be used. The CsoundUnity.cs script should be attached to the main Camera so that it will be active for the entire game. With the script attached to the main Camera it can be accessed by any of the scenes script using the GetComponent() method. The code examples below will borrow from those presented in the previous sections.

```
public class PlayerController : MonoBehaviour {

    public float speed=50;
    private Rigidbody rBody;
    private CsoundUnity csoundUnity;

    void Start()
    {
```

```
        rBody = GetComponent<Rigidbody>();
        csoundUnity = Camera.main.GetComponent<CsoundUnity>();
    }

    // Update is called once per frame
    void FixedUpdate ()
    {
        float moveHorizontal = Input.GetAxis ("Horizontal");
        float moveVertical = Input.GetAxis ("Vertical");
        Vector3 movement = new Vector3 (moveHorizontal, 0.0f, moveVertical);
        rBody.AddForce(movement * speed);
    }
}
```

Once a reference to the object has been created, all of its public member functions are available to call. The easiest way to trigger a sound in a game is by sending a score statement to a Csound instrument using CsoundUnity's sendScoreEvent() method. Let's assume that the code given below is taken from the .csd file passed to the CsoundUnity script.

```
<CsoundSynthesizer>
<CsOptions>
-odac -b64
</CsOptions>
<CsInstruments>
sr     =     44100
ksmps     =     32
nchnls     =     2
0dbfs    =    1

instr 1
aEnv expon p4, p3, 0.01
aOut oscili aEnv, p5
outs aOut, aOut
endin

</CsInstruments>
<CsScore>
f0 [60*60*24*7]
</CsScore>
</CsoundSynthesizer>
```

Instrument 1, when triggered, will output a simple sine wave with an exponential envelope. The *f0 [60*60*24*7]* score statement tells Csound to run for one week uninterrupted while it listens for realtime events. In order to trigger the instrument to play we can send a simple score statement. A good place to trigger a sound would be when a collision takes place.

```
using UnityEngine;
using System.Collections;

public class PlayerController : MonoBehaviour
{
    (...)

    void OnCollisionEnter(Collision other)
    {
```

```
        if(other.gameObject.name=="Wall")
            csoundUnity.sendScoreEvent("i1 0 1 1 200");
    }

    (...)
}
```

Each time the player object hits a wall it will trigger Csound to play instrument 1 for 1 second at full amplitude, with a frequency of 200 Hz. It would also be quite easy to use the magnitude of the collision to control the velocity of the sound. The Collision reference passed to the OnCollisionEnter() method contains information about the collision which can be used to attain the magnitude of the collision.

```
    void OnCollisionEnter(Collision other)
    {
        if(other.gameObject.name=="Wall")
        {
            float map = collision.relativeVelocity.magnitude;
            csoundUnity.sendScoreEvent("i1 0 1 "+amp.toString()+"1 200");
        }
    }
```

2.3.1 Controlling sounds using named channels

Instruments that are already playing in Csound can be controlled in real-time using named channels. Unique software buses can be created that can be used to share information between Unity and Csound. Unlike the score statement system presented above, channels allow control over instruments in real-time. To make this example a little more interesting, some simple modulation has been added to the previous Csound instrument. Also note that the instrument is started directly from the section.

```
<CsoundSynthesizer>
<CsOptions>
-odac -b64
</CsOptions>
<CsInstruments>
sr     =     44100
ksmps     =     32
nchnls     =     2
0dbfs    =    1

instr 1
kX chnget "xPos"
kZ chnget "zPos"
aMod oscili 1, kX*10
aOut oscili aMod, kZ*100
outs aOut, aOut
endin

</CsInstruments>
<CsScore>
i1 [60*60*24*7]
</CsScore>
</CsoundSynthesizer>
```

When the above instrument is started, it will constantly look up the values stored in the two named channels, "xPos" and "yPos". Values can be sent from Unity to these channels using CsoundUnity's setChannel() method. The player's X and Y position are

potentially updated with every frame, therefore it is best to send the player's position to Csound during the Update() method.

```
    // Update is called once per frame
    void FixedUpdate ()
    {
        float moveHorizontal = Input.GetAxis ("Horizontal");
        float moveVertical = Input.GetAxis ("Vertical");
        Vector3 movement = new Vector3 (moveHorizontal, 0.0f, moveVertical);
        rBody.AddForce(movement * speed);

        csoundUnity.setChannel("xPos", gameObject.transform.position.x);
        csoundUnity.setChannel("zPos", gameObject.transform.position.z);
    }
```

The player's on-screen position will now control the timbre of the sound being produced with Csound.

2.3.3 Triggering one-shot samples

Triggering samples is a staple technique in sound design for games. CsoundUnity ships with some simple methods for playing back audio files. Any files that are needed for playback should be placed in the Assets/Audio folder. If one does not exist, create one. The call to sendScoreEvent() in the previous OnCollision method can be replaced with a called to CsoundUnity's playOneShot() method.

```
    void OnCollisionEnter(Collision other)
    {
        if(other.gameObject.name=="Wall")
        {
            csoundUnity.playOneShot("explosion.wav");
        }
    }
```

2.3.3 Working with audio files

Users wishing for more control over their audio files may prefer to use the audioLoad() method.

```
void CsoundUnity.audioLoad(string filename, string ID, bool
startPlaying = false, float volume=1)
```

This method prepares an audio file ready for playback. It can be used to instantiate an audio file for playback at a later stage, or can be set up to start playback immediately if startPlaying is true. The string passed as ID can be used to control the audio file in other audio methods.

This method should always be called in the script's Start() method once the CsoundUnity object has been accessed in the script's Awake() function.

```
    void Start()
    {
        csoundUnity.audioLoad("loop_1.wav", "loop1", true, 1);
    }
```

Once this method has been called, users can control any of their audio files with a host of different utility methods such as audioPlay(), audioSend(), audioVolume(),

audioCrossFade(), audioSpeed(), etc. Each of these utility functions is passed the same string ID that was used when audioLoad() was called.
Further details on these methods may be found in the CsoundUnity documentation.

Conclusion

Csound is a powerful audio engine that can be employed in many different ways. From basic synthesis to complex generative algorithms, it provides an excellent toolkit for sound designers. CsoundUnity attempts to bring all the power of Csound to the Unity game engine.

In this text we have covered the basic core principals of Unity. We have seen how collisions work, how scripts can be attached and used in connection with GameObjects, how cameras work, and how to implement simple character movement. We have also seen how CsoundUnity can be imported to a scene in a game, and how the CsoundUnity script can be used therein.

The simple utility classes provides by CsoundUnity are included for two reason. 1) so that sound designers with no Csound knowledge can start using it without having to learn Csound and 2) so that users can see how to add C# implementation methods for different Csound API methods. However, these methods don't get anywhere near the level of control that can be achieved by writing custom solutions in the Csound language itself.

While most audio middle-ware provides methods for audio file playback, very few provide synthesis engines for the generation, manipulation, and creation of sounds on the fly. It is for this reason alone that CsoundUnity presents a quantum leap forward in audio middle-ware for games.

Website: https://github.com/rorywalsh/CsoundUnity