SONIFYING GAME-SPACE CHOREOGRAPHIES WITH UDKOSC

Robert Hamilton
Center for Computer Research in Music and Acoustics (CCRMA)
Stanford University
rob@ccrma.stanford.edu

Figure 1: A UDKOSC driven player character in an OSC-controlled dive.

ABSTRACT

With a nod towards digital puppetry and game-based film genres such as machinima, recent additions to UDKOSC offer an Open Sound Control (OSC) input layer for external control over both third-person "pawn" entities, first-person "player" actors and camera controllers in fully rendered game-space. Real-time OSC input, driven by algorithmic process or parsed from a human-readable timed scripting syntax allows users to shape intricate choreographies of timed gesture, in this case actor motion and action, as well as an audiences' view into a game-space environment. As UDKOSC outputs real-time coordinate and action data generated by UDK pawns and players with OSC, individual as well as aggregate virtual actor gestures and motion can be leveraged as drivers for both creative and procedural/adaptive gaming music and audio concerns.

Keywords

procedural music, procedural audio, interactive sonification, game music, Open Sound Control

NIME'13, May 27 – 30, 2013, KAIST, Daejeon, Korea.

1. INTRODUCTION

Virtual environments from gaming engines have been repurposed by various artistic communities for uses ranging from digital filmmaking to interactive networked musical performance [4]. Digital actors controlled in real-time by individual performers interacting over networks require much the same rehearsal efforts and ensemble choreographies necessary to coordinate actors and dancers in similar "real-world" performances. While artificially intelligent or "AI" units capable of reactive motion and action are common in many gaming experiences, external algorithmic or scripted control of game-space actors are not standard components of existent gaming toolsets.

The use of OSC [11] commands as a control system for game engines allows users the flexibility to create direct causal relationships between external data or control sources and intricate in-game motion and action. The OSC implementation for UDKOSC affords users the choice of detailed scripting tools (such as the included OSCControl application) or custom algorithmic control systems of their own design. In short, any data source that can be abstracted into a series of Open Sound Control messages can be used as input to the system.

Complex relationships between multiple game-entities can be tested, prototyped and ultimately driven by any OSC-compatible language. For a project like UDKOSC which already exports game-data over OSC for external processing and analysis, the proverbial loop has been closed, allowing completely autonomous control over game motion and action by any algorithmic or process-based system.

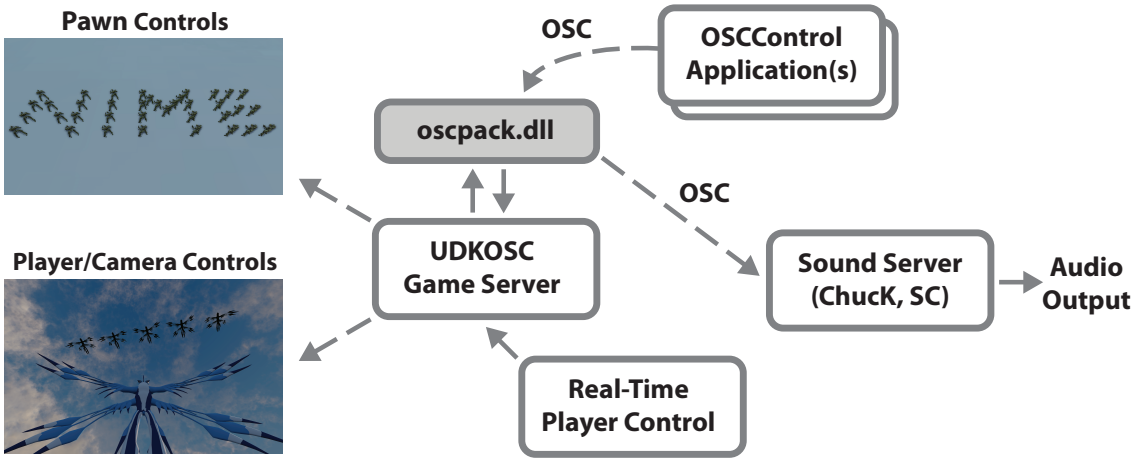

Figure 2: Flow

2. UDKOSC

Introduced in 2011, UDKOSC[1] is an open-source game-type modification to the Unreal Development Kit (UDK)[2] gaming engine featuring OSC input and output streams for the controlling, tracking and sonification of in-game actions and motions [5]. Customized game functionalities built into UDKOSC are applicable to modalities including real-time musical performance and composition, dynamic local and networked performance, procedural music/audio and rapid-prototyping of interactive game-sound design.

UDKOSC was designed to support the creation of immersive mixed-reality musical performance spaces as well as to serve as a rapid prototyping workflow tool for procedural/adaptive game audio professionals [10][7]. Data points tracked in UDKOSC include actor, projectile and static mesh coordinate positions in world-space, in-game events such as collision, and real-time ray tracing from player actors to identify interaction with specific object types and classes.

UDKOSC is a set of UDK custom class files, which when compiled, take the form of a custom game-type within the UDK. When users launch a UDKOSC game instance, a customized oscpack Windows .dll is bound to the game engine using Unrealscript's DLLBind functionality. From the UDK commandline, Users can then instantiate an oscpack UDP connection, assign a target hostname and port, and toggle on or off OSC input and output streams.

As seen in Figure 2, OSC input to UDKOSC is read via oscpack and mapped to a series of data structures that are shared between the Windows C++ .dll and the custom UDKOSC class files. OSC output from UDKOSC is similarly passed back to oscpack and pushed out over UDP to the defined target hostname and port. And while most current uses of UDKOSC connect a sound-generating server to the OSC output stream, that data can in theory be used for any analytical or archiving purposes as well.

Input data for the control of game pawn, player or camera actors is designed to offer detail and flexibility in the manner in which actors are moved and controlled within the game engine. OSC input can be used to drive actor velocity and rotation in three-dimensions, and can be targeted towards numerous entities individually or simultaeously through the use of OSC bundles. In this way, detailed choreographed positioning and action can be carried out, from Figure 2's spelling of "NIME" with 37 UDKOSC Pawns, to the "flocking" actions of bird actors, to intricate camera motion and cut scenes.

A more complete description of UDKOSC and a series of past multi-modal musical project implementations featuring UDKOSC can be found in [6]. A detailed development history of the project can also be found on the CCRMA UDKOSC Wiki page[3].

3. RELATED WORK

The control of visual elements using OSC messages can be found in a number of interactive multi-media projects. In the Processing language, OSC control is made possible through the use of user libraries such as Andreas Schlegel's oscP5 library[8]. Similarly, in the Unity game engine, Jorge Garcia's UnityOSC project allows for bidirectional OSC communication [3]. The control of virtual gesture specifically for the synthesis of musical sound has been investigated by Bouënard et. al with an eye towards the capture and understanding of natural performative gestures [1]. And Steven Sinclair's work with DIMPLE looked to the manipulation of virtual objects using OSC and their resultant physics-based behaviors as musical controllers [9]. The use of commercial gaming platforms for musical sonification has also recently been explored by Cerqueira et al. in their *Soundcraft* performance piece, wherein an online battle within the Starcraft 2 video game serves as control data for a multi-channel musical work, sonified in ChucK[2].

4. VIRTUAL GESTURE

Within the scope of this project, virtual gesture can be defined as any series of actions or motions performed by one or more game-space actors within a given time frame. For instance, a dance-like series of motions ("actor sprints up a ramp, jumps, twirls and lands") or a more direct mapping of human-physical gesture via a Microsoft Kinect controller ("actor's skeletal mesh mimicks user swinging an arm side-to-side") are each considered a "gesture" in this context. Similarly, as in human dance, fully choreographed group "gestures" can be comprised of multiple actors moving in a coordinated fashion.

In the recent UDKOSC work *ECHO::Canyon*, which tracked the flight and interactions between a "Valkordia", a bird-like

[1] `https://github.com/robertkhamilton/udkosc`
[2] `http://www.udk.com`
[3] `https://ccrma.stanford.edu/wiki/UDKOSC`

creature, and its environment, gestures idomatic to flight were used, such as diving, swooping, and flocking (with OSC-controlled pawn units) [4]. Similarly, in the 2010 UDKOSC work *Tele-harmonium*, the primary character gestures focused on a robot actor's motion around an environment as well as the flight and homing patterns of glowing projectiles.

5. OSCCONTROL

OSCControl is a small ruby application that translates sets of human-readable control commands into OSC messages or bundles, complete with accurate timing information. OSCControl makes use of the osc-ruby[5] Ruby gem for OSC message and bundle generation. Users can create scripts controlling both pawn and camera movements. A series of scripted commands are processed by OSCControl, formatting them as valid timed OSC messages, and sending them to the desired IP and PORT of a running UDKOSC server. Slewed parameter values over specific time intervals are easy to create, as are batched commands, sent as OSC bundles.

OSCControl currently creates a time-ordered array of valid OSC messages and bundles and subsequently streams entire control scripts serially, meaning any timed commands written in the script will be sent in an ordered fashion. So at each execution of OSCControl, the generated batch of OSC messages and bundles are streamed out to the desired target host and port in real-time.

OSCControl scripting commands are formatted in a human-readable format that allows for single-line description of slewed values. There are two primary classes of scripting commands, those that control instances of the UDK OSCPawn class and those that control instances of the default Camera class.

5.1 Actor Controls

In UDKOSC there are currently three types of Actors: human-controlled "players", script controlled "pawns" and game AI-controlled "bots". "Player" and "Pawn" commands start respectively with "playermove" or "pawnmove", and are followed by a series of parameters, their associated target values, an optional "slew" time, and a userid number.

```
pawnmove <action> <target> <slew> <player-id>
```

For example, a simple command to immediately set a player's speed to 4000 (approximately 10x the default UDK pawn speed) would use:

```
playermove speed 4000.0 1
```

Here, "4000.0" is the value of the player's speed parameter. As there is no "slew" time associated with this message, the speed change will happen instantly, resulting in an OSC message that looks like:

```
/udkosc/script/playermove/speed 4000.0 1
```

We can make this call more interesting by interpolating an actors speed from its current value (stored in OSCControl) to the target value ("2000.0" in this example) over a period of time, represented in milli-seconds. It should be noted that while OSCControl's default temporal step-size is 20 ms, that value can be adjusted as necessary.

```
playermove speed 2000.0 200.0 1
```

```
/udkosc/script/playermove/speed 200.000000 1
/udkosc/script/playermove/speed 400.000000 1
/udkosc/script/playermove/speed 600.000000 1
/udkosc/script/playermove/speed 800.000000 1
/udkosc/script/playermove/speed 1000.000000 1
/udkosc/script/playermove/speed 1200.000000 1
/udkosc/script/playermove/speed 1400.000000 1
/udkosc/script/playermove/speed 1600.000000 1
/udkosc/script/playermove/speed 1800.000000 1
/udkosc/script/playermove/speed 2000.000000 1
```

So over the slew period of 200.0 ms, we ramp the player's speed parameter from its current value (here a value of 0.0) to the target value of 2000.0.

Actors - either players or pawns - can be controlled using the following commands. Some commands can be used with slew timings while others, which typically control single discrete events like "jump" or "crouch" cannot be used with slew timings. A description of each control follows:

```
pawnmove x <degree-value> <slew-time> <userid>
pawnmove y <degree-value> <slew-time> <userid>
pawnmove z <degree-value> <slew-time> <userid>
playermove pitch <degree-value> <slew-time> <userid>
playermove yaw <degree-value> <slew-time> <userid>
playermove roll <degree-value> <slew-time> <userid>
pawnmove speed <value> <slew-time> <userid>
playermove jump <height-value> <userid>
pawnmove teleport <x> <y> <z> <userid>
playermove stop <userid>
```

To move an actor in a specific direction, the "playermove" or "pawnmove" commands with X, Y, and Z coordinates are used to set the actor's direction vector with a degree value relative to the world's absolute coordinate grid. If the user's speed is set to be non-0, setting the X and Y coordinates will start the user moving in the desired direction. As the Z coordinate represents the vertical plane, Z coordinate motion will only cause effect if the user is currently in a flying state.

5.1.1 Stop Control

To stop an actor's motion, we use the "playermove stop <userid>" command. This will stop the actor moving. No value needs to be sent with a stop command and after a stop command is sent, the next playermove x, y, z, or jump command will toggle the stop state off, allowing the user to move freely.

5.1.2 Speed Control

Actor speed can be set using "playermove speed <value> <ms-slew> <userid>". The UDK default speed is 300-400 in UnrealScript. A speed of "0" will not allow Actors to move in any direction. Speed values can be slewed, to create accelerations or decelerations.

5.1.3 Jump Control

An Actor can be made to jump by sending a "playermove jump <height-value> <userid>" message. The height to which the user will jump is sent as the value for the jump message.

5.1.4 Teleport Control

Actors can be moved to any coordinate location in the current environment instantly using the teleport command. This is useful in starting actor motions and actions from a specific location.

[4] http://www.ustream.tv/recorded/31968509
[5] https://github.com/aberant/osc-ruby

6. CAMERA CONTROLS

The Camera associated with an actor can be controlled independently using the following commands. Note that camera controls are formatted in much the same way as player controls except that there is currently no "cameraid" in use:

```
cameramove x <degree-value> <ms-slew>
cameramove y <degree-value> <ms-slew>
cameramove z <degree-value> <ms-slew>
cameramove pitch <degree-value> <ms-slew>
cameramove yaw <degree-value> <ms-slew>
cameramove roll <degree-value> <ms-slew>
```

7. APPLICATIONS

External control of in-game actors has applications for both research and artistic purposes.

7.1 Research Applications

The ability to script specific sets of actor motions and actions affords researchers a recreatable dynamic virtual gesture set that can itself be interacted with by autonomous subjects operating within the virtual environment. As part of an ongoing study into the perceptual coherence of sonified game-space gesture, UDKOSC gesture scripting is already in use, creating gestures for musical sonification.

7.2 Artistic Applications

One ongoing artistic direction with UDKOSC gestural scripting is the creation of choreographies controlling multiple actors, much along the lines of a classic Hollywood musical chorus line. Such complex multi-user choreographies have proven difficult to coordinate by human users working with manual game control systems. Through scripting, we can create tightly synchronized formations and coordinated actions and map them to musical structures.

7.2.1 ECHO::Canyon

The first musical work to make use of UDKOSC's new scripting controls was *ECHO::Canyon* by composer Robert Hamilton and visual artist Chris Platz, an interactive audio-visual piece tracking the motion of a flying character through a rendered environment. During the work, ECHO::Canyon makes full use of UDKOSC's player, pawn and camera control systems, moving between OSC control and manual control at various times in the piece.

The work begins with the player character, a bird-like creature called a Valkordia (see Figure Figure 3) sitting atop a mountain top. OSC output calculating the player's distance from a large crystal is used to drive multi-channel sound synthesis. In the opening scene, the camera is detached from the pawn and moves slowly across the game world, eventually catching up to the player character and attaching itself to the character's viewpoint. At this point, the pawn launches itself into the air, and begins a flight sequence controlled by OSC, skimming the ground (driving another synthesis process), and eventually flying to a determined location, where control over the player is relinquished to the live performer.

During these choreographed sequences, OSC has also been controlling the flight paths of a flock of birds, whose height over the landscape is being used to drive yet another synthesis process. The rich soundscapes generated by these interactions serve as a strong showcase for the potential of future scripted control examples made with UDKOSC.

Figure 3: A Valkordia pawn controlled by OSCControl in the work *ECHO::Canyon*, with a sounding crystal visible on the mountain peak.

8. REFERENCES

[1] A. Bouënard, M. Wanderley, and S. Gibet. Gesture control of sound synthesis: Analysis and classification of percussion gestures. *Acta Acustica United With Acustica*, 96(4):668–677, 2010.

[2] M. Cerqueira, S. Salazar, and G. Wang. Soundcraft: Transducing starcraft 2. In *Proceedings of the New Interfaces for Musical Expression Conference*, Daejeon, Korea, 2013. NIME.

[3] J. Garcia. Unityosc github repository. `https://github.com/jorgegarcia/UnityOSC`. Accessed: 2013-04-26.

[4] R. Hamilton. q3osc: or how i learned to stop worrying and love the game. In *Proceedings of the International Computer Music Association Conference*, Belfast, Ireland, 2008. ICMA.

[5] R. Hamilton. Udkosc: An immersive musical environment. In *Proceedings of the International Computer Music Association Conference*, Huddersfield, UK, 2011. ICMA.

[6] R. Hamilton, J.-P. Caceres, C. Nanou, and C. Platz. Multi-modal musical environments for mixed-reality performnace. *Journal for Multimodal User Interfaces (JMUI)*, 4:147–156, 2011.

[7] L. J. Paul. Video game audio prototyping with pure data.

[8] A. Schlegel. oscp5: An implementation of the osc protocol for processing. `http://www.sojamo.de/libraries/oscP5/`, 2011. Accessed: 2013-04-26.

[9] S. Sinclair and M. M. Wanderley. A run-time programmable simulator to enable multi-modal interaction with rigid-body systems. *Interact. Comput.*, 21(1-2):54–63, Jan. 2009.

[10] C. Verron and G. Drettakis. Procedural audio modeling for particle-based environmental effects. In *Proceedings of the 133rd AES Convention*. AES, 2012.

[11] M. Wright and A. Freed. Open sound control: A new protocol for communicating with sound synthesizers. In *Proceedings of the International Computer Music Association Conference*, pages 101–104, San Francisco, USA, 1997. ICMA.