COSC 470

Final Report

Kevin Shen

Supervisor: Geoff Wyvill

October 2012

**Contents**

1. Abstract
2. Introduction to Virtual 3D Sculpting
3. Playstation Move controller and Move.Me software
4. Work Done (is there a better phrase for this? Project specifics?)

4.1 Selection and set up of tools

4.2 Launching the application and sculpting with the Move controller

4.3 Rendering the sculpture

4.4 Visual cues

1. Discussion
2. Future work
3. Summary
4. References

**1. Abstract**

The objective of this project was to create a virtual sculpting tool that uses 3D input. The application uses the Playstation Move controller as a physical input device and the Move.Me software to obtain 3D coordinates of the controller. The Move.Me software is run on the Playstation and supplies 3D coordinates via wireless connection to the client, which renders the sculpture as a trail of spheres. A 3D input system eliminates problems inherent in traditional computer sculpting tools, such as converting 2D input to 3D coordinates. The application uses several forms of visual cues to enforce the illusion of three dimensional sculpture and aid in the user’s hand-screen coordination. I demonstrate that 3D sculpting applications may become more intuitive and have fewer limitations with true 3D input.

**2. Introduction to Virtual 3D Sculpting**

This will discuss other research projects involving 3D sculpting, including Ross’s project and its limitations (helix problem). Also discuss the previously high cost of motion tracking systems versus the relative low cost of consumer products like the PS3.

**3. Playstation Move controller and Move.Me software**

Sony markets a relatively cheap motion-tracking system for use with their Playstation3 video game console. The system consists of the Move controller, the Eye camera, and Move.Me software that creates a server on the Playstation. The controller features a coloured ball at one end, which is used by the Eye camera to track the position of the controller. The controller also has a trigger and six buttons. See Figure 1 for the structure and button layout of the Move controller.



Figure 1: Playstation Move controller

The software is capable of reporting six degrees of freedom X, Y, and Z coordinates relative to the Eye camera as well as three angles representing orientation of the controller. The software communicates with a client via a local wireless network. Data is represented as multiple states. This includes, among others, state of the controller, camera, and current camera frame. Button presses and changes in Move.Me are included in the controller state.

**4. Work done**

**4.1 Selection and setup of tools**

The current market offers many consumer motion-tracking systems. Most are associated with current-generation game consoles; each of the Wii, Xbox 360, and Playstation3 has motion-tracking peripherals. I chose to use the Playstation’s Move controller because of its availability and precision. The Otago Graphics Research Group was already in possession of a Playstation3, Move controller, and the Move.Me software. The motion data reported by the Move.Me is specific to the controller, whereas the Xbox’s Kinect motion tracking system tracks motion of the entire body, resulting in less precise reports with unnecessary data. The Wii has no published support for developers wishing to track motion of the controller. Thus, the Playstation Move was the best choice for this application.

The Move.Me software is supported for Linux and Windows in C and C#. I intended to use the C libraries in a C++ project on Linux, but encountered difficulties with interleaving C and C++ code. The Linux workstation also lacked appropriate wireless drivers. Finally, the C# Move library is able to easily connect to the Playstation server whereas the C library is not. Given the project’s time constraint of one semester, I switched to a C# environment on Windows to gain access to the more established C# library. Once the project platform and language were chosen, a connection for data transfer between the client and Playstation3 was created with a wireless router.

**4.2 Launching the application and sculpting with the Move controller**

The Playstation must be on and running the Move.Me software in order for the client to work.

The sculpture application is built on top of the GUI provided by the Move.Me library. At the launch screen, the user must first connect to the Playstation server. Enter the IP address and port number of the Playstation server into the fields in the upper left corner of the GUI (see Figure 2). The IP address and port can be found in the upper left corner of the Playstation display (see Figure 3). Establishing the connection will result in immediate continuous transfer of controller data from the Playstation server to the client.

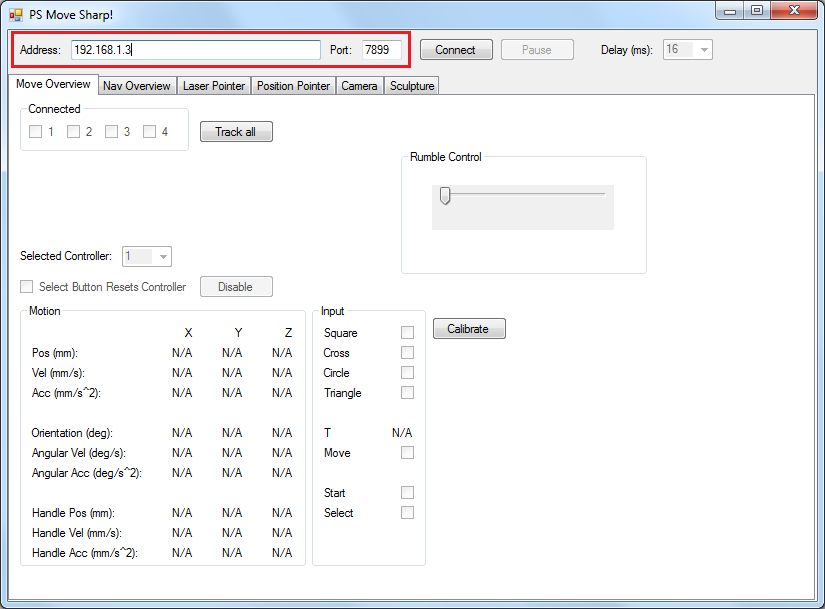


Figure 2: The application launch screen with server information fields highlighted



Figure 3: Playstation Move.Me display

To begin sculpting, click on the “Sculpture” tab. The application will launch a fullscreen window depicting a museum scene with a turntable, as seen in Figure 4. Once in full screen, sculpting can be controlled using only the Move controller. The application is calibrated such that the user should stand approximately 1.5 metres from the Eye camera for a full range of motion. A coloured sphere indicates the position of the sculpting tool in virtual space.

To create or add to the sculpture, pull the trigger and move the controller.

To rotate the turntable and the sculpture clockwise, press the Cross button on the controller.

To rotate anticlockwise, press the Circle button.

To delete the current sculpture, press the Triangle button on the controller.

To switch between the full screen window and the GUI, press the Square button.



Figure 4: The empty sculpture scene

**4.3 Rendering the sculpture**

The sculpture is rendered as a series of spheres. The position of each sphere is stored in a list. Each time the state of the controller is updated, the application compares the current position of the sphere to the location of the last sphere that was added to the list. If the two spheres are greater than a certain distance apart and if the controller state indicates the trigger is being pressed, the sphere is added to the list of spheres to render.

Because the Move.Me software reports the 3D coordinates of the controller relative to the Eye camera, almost no calculation is needed to determine where to render the sphere on the screen. Given the Eye camera exists at position (0, 0, 0), the detectable range of motion for the controller is between -700 and 700 on the X axis, -450 and 450 on the Y axis, and 100 and 1800 on the Z axis. See Figure 5 for reference. Note these X and Y limitations when the user stands approximately 1.5 metres from the Eye camera; because the Eye camera’s vision is limited to a frustrum, the X and Y coordinates that bound the area of detectable motion vary based on the Z coordinate.

The application simply establishes an OpenGL scene with the camera at approximately the same location as the user’s eyes. The centre of the workstation screen is considered (0, 0, 0) because the avatar of the sculpting tool cannot extend out of the screen (in the negative Z direction). The height of the user’s eyes above the centre of the workstation screen depends on both the height of the workstation desk and the height of the user; as this is variable, the application is currently calibrated using a workstation screen height of 1m and user of 175 cm.

The arbitrary units reported by the Move.Me software are the units used by the OpenGL scene. By experimentation, one centimetre is equal to seven arbitrary units. Given the user stands 1.5 metres from the screen and his or her eyes are 30 centimetres above the screen midpoint, it is a simple conversion to place the OpenGL camera at (0, 210, -1050).

With the camera appropriately placed, it is a simple matter to render the spheres at the position reported by the Move.Me software.



Figure 5: Coordinate system of the scene

**4.4 Visual cues**

A number of visual cues are used to provide reference for position and enforce the illusion of 3D.

The backdrop of the scene provides context for the 3D position of the sphere. Grid lines on the floor and turntable give the illusion of perspective and help the user track the X and Z position of the avatar. The backdrop was created by Ross Phillips for his Octagon project [1].

The position of the drop shadow under the avatar’s sphere helps the user understand where the avatar is in the X and Z directions. The size of the shadow, which grows as the avatar approaches the ground, provides context for the height of the avatar.

Finally, the rotation function allows the user to examine the sculpture from any angle. This allows the user to better understand the position of each sphere relative to all the other spheres.

**5. Discussion**

The use of position tracking software in this project eliminated ambiguities associated with traditional 3D modelling and sculpting programs that use 2D input. In these applications, a mouse click with 2D screen coordinates in X and Y must be converted to a set of 3D world coordinates in X, Y, and Z. One way to do this is by treating the screen as a plane called the view plane, which exists in 3D space in front of the scene’s camera. A ray is cast from the camera and travels through the coordinate of the clicked point on the view plane. The ray continues on through to the scene and may intersect existing 3D objects in the plane.

This method is suitable for tasks such as selection of visible objects. The first object that intersects the ray is the object that is closest to the camera and thus the closest visible object. However, ray casting is a poor method for object creation. In a blank scene, it is utterly ambiguous which 3D world coordinate the user intended to indicate. Any coordinate on the ray between the view plane and the end of the scene is valid. Thus, an object cannot be created with a single click, because 3D coordinates cannot be determined from a 2D mouse click coordinate without additional information.

Instead, current 2D-input 3D modelling applications such as Maya default to object creation at the origin. The user can specify 3D coordinates at which to create basic 3D shapes (primitives) by manually entering world coordinates. This method of creation takes 8 mouse clicks and keystrokes in Maya. Because this method is unwieldy, users often choose to create primitives at the origin and then scale, rotate, and translate the primitive as necessary. Creating a unit cube and translating it along 3 axes takes 6 mouse clicks in Maya.

The Move system provides 3D coordinates of the controller to the client. With true 3D input, creation of primitives at a specific world location can be done with a single button press. However, users of 3D input must be able to coordinate their real position with screen position. This is done with the use of an avatar; the controller’s position is represented on the screen as a differently-coloured sphere. The user should be able to move the physical controller and see the avatar move as they expect on the screen.

For movement to be intuitive, the illusion of 3D must be enforced. The primary way of doing this was with the background. I initially intended to mirror the real world in the virtual world by using video frames from the Eye camera as the background. This failed because the video frames provided by the Move.Me software were too low-resolution to give any meaningful visual cues and took too long to render in GL despite their resolution. Instead, I used the backdrop from Ross Phillips’s Octagon project [1].

With a static background the avatar can either mirror or emulate the user’s movements. In mirrored movement, the avatar grows as the controller gets closer to the screen, as it would in a real mirror. Emulated movement is a direct mapping of movement from the real to virtual world; in emulated movement, the avatar shrinks as the controller moves closer to the screen.

What to say about why I chose mirrored? It felt more natural and was the only option when I first intended to use video feed as the background. Need concrete reasons, but no test group like Lyn.

**6. Future work**

Though the application is relatively intuitive, new users still require instruction. Rotation could be made more intuitive by moving the avatar to hooks or latches on the edge of the turntable to toggle rotation and allowing rotation to be controlled by gestures of the controller.

The sculpting function itself is limited. Many features in Ross Phillips’s Octagon project could be implemented to increase the range of tools available to the user [1]. Sculpting is currently purely additive. Allowing for subtractive sculpting by deletion of primitives would improve the precision with which the user could create 3D forms. Greater flexibility with primitives could also be implemented; allowing creation of additional primitives such as cylinders and cubes, as well as allowing the user to adjust the size of the primitive being created, would increase precision. Allowing users to select colours by moving the avatar to paint buckets would be highly intuitive.

The current calibration is hard-coded for a certain user distance and height. Allowing the user to calibrate when the application is launched would make movement more accurate and intuitive. This could be accomplished by having the user register the controller position when it is in front of their eyes to determine the OpenGL camera’s position. The sculpting area could also be defined by the user to calibrate sensitivity of movement.

**7. Summary**

\*\* Account for time necessary to select and set up tools

\*\* Account for unexpected delays – switching platforms

\*\* Calibration is key and can make an application unusable, even if the mechanics are sound

**8. References**

[1] Phillips, R. 2011. *Cooperative Sculpture*. Technical Report. University of Otago.