CSC494 AR Sandbox Project Final Report

Written by: Philip Smith

Abstract

This is a project under KMDI. The AR Sandbox is an open source software developed by UC Davis. It's a simulation that takes the depth image of a sand surface and projects that color mapping back on the sand in real time. The goal of this project was to gain a better understanding of the AR Sandbox code and to set the foundation for further additions to the simulation. What was accomplished was a 'bouncing ball' animation based on a design by Chua Hock-Chuan in 2012. A 3D sphere was drawn into the scene, and using translations, traveled around the display on a two-dimensional plane above the sand model. When it reaches a certain threshold, it changes direction and continues moving, simulating the act of it reflecting or 'bouncing' off.

The work done here leaves room for improvement and further experimentation, but with the knowledge gained and the steps taken, those who come later will have an easier time getting acclimatized to the nature of the code. The animation achieved here can be the basis for a variety of visual effects in the simulation to make it a more interesting activity for users. In addition, through debugging, the concept of a treasure hunt activity being built into the sandbox was brought to fruition. But by far the most difficult idea is to project a 3D model on the sand surface itself, as this would require an implementation of ray casting and calculating bounding boxes, which for something like the model of the AR Sandbox is no easy feat. Due to time constraints, a couple issues had to be overlooked in favor of what were more pressing concerns at the time. Because of this, they are also available to tackle in the future. These dealt with the bounds of the ball animation and maintaining a centered view on the animation in 3D space.

Section I: Introduction

This course work was conducted under the Knowledge Media Design Institute (KMDI) at the University of Toronto (UofT). In a previous project, KMDI had built their own Augmented Reality Sandbox (AR Sandbox), utilizing the guidelines and software publicly provided by the University of California, Davis (UC Davis). The AR Sandbox is a simulated topography model that can be interacted with and modified in real time. It accomplishes this with an XBox 360 Kinect and a projector. The Kinect is set up to consistently interpret the depth image of a fixed area of sand. The projector then projects that depth image on the sand surface, providing a visual representation of the sand's depth in relation to the Kinect on the sand itself. As a user interacts with the sand and modifies its shape, the depth image, and thus the projection, changes. This is all ran using the VR Toolkit 'Vrui', also developed at UC Davis.

While the guidelines UC Davis provides effectively allow the installation and use of the AR Sandbox, they do not provide a clear explanation of the code itself. The project's code is extensive, with many moving parts that ultimately result in the AR Sandbox simulation. As such, if one wanted to implement their own additions to the code, it is not immediately obvious where they would start. An official forum is available for users to post questions and answers relating to the program and hardware (Kreylos, 2016). However, many of the discussions and questions revolve around the initial set up and rarely touch upon complicated code augmentations. Because Vrui is not widely used outside of UC Davis, the only source of aid for tackling it is the provided documentation. Which, while helpful, can also be improved in its accessibility.

The task of this course work was to investigate the code and gain enough of an understanding so that it was possible to contribute a meaningful addition to the sandbox simulation. This required mapping out the flow of data through the program and understanding where the final output was being relegated.

Section II: The Process

Over the course of the project, it was determined that managing to get a 3D object in the simulation and applying some degree of animation would provide a substantial building block for any future endeavors. The ultimate goal is to further expand the AR Sandbox to even more interactive capabilities. This may include virtual characters that exhibit certain behaviours based on their position on the height map, or perhaps fully developed games with objectives and rules. This will be touched upon later on in the report.

In order to build my understanding of the code base, I made use of a variety of tools. Early on in the project, the open-source software Sourcetrail helped greatly in providing a better understanding of the code's flow. It constructed a visual map of dependencies between the files, which allowed me to follow where the data was being manipulated and outputted. In addition, it led to the discovery of the functionality of the Vrui toolkit and how it contributed to the AR Sandbox display.

Following the discovery of Vrui's functionality and its use of the OpenGL API, I was able to begin developing my code. This went through a few stages, as I had to first become acclimatized to OpenGL, specifically when it came to simple animations. I came upon a tutorial that helped guide the direction of the project moving forward. It was written by Chua Hock-Chuan in July, 2012. What they did was construct a circle in 2D space and applied a 'bouncing' animation to it. I determined this was suitable as a basic animation to experiment within the AR Sandbox's display. The idea was that by starting off simple, I can focus on understanding Vrui and OpenGL in the context of the AR Sandbox rather than allocating the majority of my time to deconstructing a complicated animation and model that had nothing to do with the AR Sandbox.

Carrying forward the basic idea of this animation, I began looking more into Vrui. Through the documentation provided by UC Davis for the Vrui Toolkit, I was able to create my own 3D scenes through Vrui. This was accomplished by repurposing example programs provided to me when I downloaded the toolkit. During this process, I was able to identify the more glaring bugs and issues that can arise from a misuse of Vrui, giving me a better understanding of the toolkit and its functions going forward.

After I managed to recreate Hock-Chuan's animation in Vrui with a 3D sphere, I felt comfortable moving it to the AR sandbox code. After building the set up with UC Davis's instructions, I was able to begin testing. Through an arduous process of trial and error for the initial portion of this stage, I was able to make the sphere visible in the AR Sandbox. The main problem I had was fully grasping the built in matrices OpenGL utilizes. This also led to the realization of how the depth image behaves in Vrui, at least in relation to the viewer (the virtual camera). Because the map is scaled to always fit the window view while still being an actual 3D model in the scene, it obscures the rest of the scene behind it. Because of this, it proved difficulty starting off to determine whether or not the sphere was being drawn at all. Using what I could determine about the scene and the viewer position, I was finally able to bring the sphere in front of the depth image. From there, using what I know about rays in 3D space, I was able to fix the origin point of the sphere to always be visible.

Section III: The Code

`Sandbox.h:`

The header file for Sandbox.cpp. This file is where you will declare any functions and variables you require for Sandbox.cpp. My contributions consisted of the following:

`textureObjectId:`

This variable is of the type `GLuint`. It's an element of the `struct DataItem` under the `Sandbox` class. It's meant to hold the ID of some texture that may be applied to the ball. While this is a relic of the example programs from Vrui that I didn't end up using, I chose to include it in the final code. This way anyone in the future who may wish to add a more stylized look to the ball has the means to do so.

`displayListId:`

This variable is of the type `GLuint`. It's an element of the `struct DataItem` under the `Sandbox` class. It's meant to hold the ID of some display list. This is an important variable to include as a display list keeps track of the vertex data of the sphere. Therefore, it gets updated at initialization of the sphere 3D model as well as each time the program animates it. Without it, the same initialization of the sphere model will be written every time an update is required.

`translatedPosition:`

This is a 3 element array of the type `Scalar` (declared in `Vrui/Geometry.h`). I used this as the structure to store the current coordinates of the ball in 3D virtual space.

`centerPosition:`

This is a 3 element array of the type `Scalar`. I used this as the structure to store the starting coordinates of the ball in 3D virtual space, which by design is in the center of the screen.

`moveSpeeds:`

This is a 3 element array of the type `Scalar`. I used this as the structure to store the ball's current direction of movement in 3D virtual space. This would act as the ball's directional vector.

`BASEtranslatedPosition:`

This is a 3 element array of the type `Scalar`. I used this as the structure to store the base form of the current translation. What this means is this array would be identical to

`translatedPosition` in terms of elements if `centerPosition` was `(0, 0, 0)`. I use this as a means to maintain the ball's position in the viewport's scene should the viewer position change.

ballRadius:

This variable is a `double`. As the name implies, it is the radius for the sphere model that acts as the 'ball'.

ballHMax, ballHMin, ballVMax, ballVMin:

These four variables are of the type `GLFloat`. They represent the bounding edges that the ball will bounce off of.

Sandbox.cpp:

This file drives the Vrui 3D space, from the models to the animation. While it relies on the other files to construct the data to build those models and animations, this file is what ultimately brings them to the screen. It utilizes the built-in Vrui Application functions `frame`, `display`, and `resetNavigation`, and the GLObject function `initContext`, all of which proved valuable for the sake of the project. This is where the majority of the code for the course work went. My contributions consisted of the following:

DataItem(void) and ~DataItem(void):

These functions are the constructors and deconstructors of the `struct DataItem` respectively. It's in these functions that I instantiate and dismantle the ID objects `textureObjectId` and `displayObjectId`.

findBounds(void):

This function is original for this project. It's a simple helper function that updates `centerPosition` based on the viewer's position in 3D space. The equation used is the parametric form of a ray:

```
r(t)= <origin vector> + t*<direction vector>
```

Where `t` is an arbitrary point on the ray. Using this equation I find the center of the screen with this new viewer position. With the updated `centerPosition`, it finds the horizontal and vertical bounds based on fixed values.

Sandbox(int& argc,char**& argv):

This is the constructor for the `Sandbox` class. In this function, I assign values to `moveSpeeds`, `BASEtranslatedPosition` , and `ballRadius`.

frame(void):

This function is called exactly once for each frame spent in Vrui. Because of this, according to Vrui's documentation, this is where all code related to changing the state of the application such as animating models should take place. This should cover everything just shy of applying those changes to the model matrix. The code I wrote for the ball animation in this function covers (in order):

- Getting the time since the last frame.
- Finding the boundaries of the animation with `findBounds()`.

  Then for each of the axes x, y, and z...

- Checking if the ball exceeds that axis's boundaries
- If the ball does exceed the boundaries, changing the expected direction and position of the ball on that axis.
- Calculating the ball's new position given its movement speed, direction and previous state.

After all of that, the only task left is to apply the resulting `transPos` elements to the model matrix.

display(GLContextData& contextData):

This function can be called multiple times in a single frame depending on the number of eyes in every window running the program. Because of this, the Vrui documentation advises that no code that changes the state of the application should be written here, indicating that it should be placed in `frame()`. Instead, code that modifies the model matrix should be placed here.

The way OpenGL works with 3D models is, by default, all models act on the same model matrix. What that means is if any geometric modification is applied to one object, such as translation, it will be applied to all models in the scene. In order to separate the objects to other matrices, one needs to call `glPushMatrix()`, write whatever OpenGL translation code they want, then close it off with a call to `glPopMatrix()`. In the case of this project, I used `glTranslated()` to apply the translation from `frame()`.

resetNavigation(void)

This function is called at the start of the application and whenever the 'Reset View' button is pressed. My only addition to this function was assigning values to `translatedPosition`. I had to do this in this function rather than the constructor of

`Sandbox` because I required the viewer's starting position when the program begins. The viewer doesn't get its starting position until `resetNavigation()` is called, and this happens after the constructor has already finished.

`initContext(GLContextData& contextData):`

This function is inherited from the GLObject class. It is called once per OpenGL context in the Vrui application. Its purpose is to instantiate OpenGL application data and store it in the `contextData`, which will then be called back in `display()`. In this function, using the variable `displayListId` that was assigned in the class constructor, I create a new display list. I then make a call to `glDrawSphereIcosahedron()` from the library `GLModels.h` in order to build the sphere's vertices. Afterwards, I end the list.

Section IV: Conclusion

As was implied in Section II, I was able to get the ball bouncing on screen. I accomplished the main goal of this course work, which was to gain enough of an understanding of the AR Sandbox code to add my own significant contribution to the simulation. Overall I believe the work I've done will help future endeavors to further understand and work with the AR Sandbox.

This did come with some compromises, though. For instance, a part of Hock-Chuan's program that I was not able to replicate in the 3D space was their use of the viewport's dimensions as bounds. To recap, in `findBounds()`, I used a method based on `centerPosition` for determining the bounds that will trigger the ball to 'bounce' and change direction. However, this is not how Hock-Chuan created their bounds. Their method was a lot more intuitive.

In the original 2D version, their equivalent of `findBounds()` was `reshape()`, a function they registered as a callback handler for whenever the window resized. In `reshape()`, they made calculations concerning the new aspect ratio of the window and applied those values to the bounds. This led to the ball registering the edges of the viewport as the bounds no matter the size or shape of the window. The problem with translating this to a 3D space is the existence of the 'near' and 'far' value. In graphics, the 'near' value is the closest distance from the camera that an object will be rendered, and the 'far' value is the furthest (Brown, 2016). With the way 3D vision usually works, perspective is like a cone: the further away from the eye, the wider the field of view. Unlike 2D space, where the 'near' and 'far' values are negligible on a 2D plane, 3D space typically allows a much more substantial space between the two. And with the cone affect, the edge of vision that an object passes is not parallel to the initial 'near' field of view. Thus, when attempting to map the bounds of a 3D model like the ball according to the viewer's field of vision, it proves difficult.

While I didn't get it working, I do have some avenues that I can leave for the next attempt. To explain, `gluOrtho2D()` is a function that Hock-Chuan used in their code. It

creates a 2D orthographic viewing region in an OpenGL scene. A sibling function to this is `gluOrtho()`, which takes 'near' and 'far' values as parameters. According to the documentation, `gluOrtho2D()` is a call on `gluOrtho()`with the 'near' and 'far' values set to -1 and 1 respectively. I attempted to recreate Hock-Chan's scene using `gluOrtho()` in the 3D scene, but tests proved to not have the desired effects. The bounds unfortunately were not mapped to the viewport's edges. Due to time constraints and the need to move on to more pressing issues in the project, I had to abandon this venture. I do believe I was on the right track though, so I encourage further testing.

In the final solution there is a minor bug that I have not found a cause for. Occasionally, when the ball reaches a bound, it will stall and slowly float offscreen. I have reason to suspect this is due to how I calculate my bounds, though I cannot point to the exact issue in my code's logic. This bug, however, is minor, as every time it has happened, the program rights itself shortly thereafter and the ball is back on screen, behaving as it should. I suspect that if Hock-Chuan's bounding method is successfully replicated, this issue should go away.

An issue that arose near the end of the project was propagated by a default setting of Vrui. It turns out that there is a feature that allows the action of moving the viewport to pan the virtual scene. At this point in time, I had not integrated the ray equation into the ball's translation code, so the ball immediately disappeared from the scene when the viewport was moved. This was a huge problem since I needed to move the viewport to the projector screen in order to project in on the sand. This issue is actually what drove me to incorporate the ray equation into the ball's translation code as an attempt to circumvent the problem this feature posed. However, while the code did what it was supposed to do according to the tests' coordinates results, the ball was not following the viewport as it panned. It instead stayed locked where it was on screen, which admittedly was a step above it travelling in the opposite direction like it was doing before. Additionally, according to the coordinate results, the viewer wasn't always in the same position when I entered and exited full screen mode. So attempting to find the viewer's position on the projector screen and hardcoding the ball to appear there wasn't an option either. In the end, I had to enter the Vrui configuration file with admin privileges and switch the `panningViewport` boolean from `true` to `false`. This fixed the issue.

There is however room to improve upon this solution. As mentioned before, the depth map is fixed to the viewer. No matter how the scene is navigated, the map never changes its orientation according to the viewer. I couldn't find the code that allowed it to do that, but should someone discover where it is and then use the same logic for the ball, then the panning viewport should no longer be an issue, regardless of the status of the configuration file.

Lastly, at the very start of the project, it was hinted that this research could lead to the making of a game. As things progressed, though, it was ruled as too ambitious for the scope of this course work. However, now that the code exists for objects outside of the depth map to enter the scene and animate separate to it, the tools are there to start on those projects. As it is, any model and animation made with OpenGL can be placed in the scene. Birds can be animated flying overhead for example. With the right animations and models, that is now completely possible in the AR Sandbox. Relating to this, I had made some attempt during the early stages to make the ball's translations more sporadic and random rather than straight paths. I did

accomplish this to an extent, though I shelved the idea and kept the ball on a straight path mainly for the sake of perfecting those simpler movements in relation to the viewer first.

One idea that I came upon by accident was that of a treasure hunt game. As was mentioned before, the depth image is a 3D model, meaning it's not just the colors that are changing. The slopes and pits are rendered in 3D in Vrui. When I was attempting to find the ball initially, I noticed that it could be partially obscured by the depth map, and that I could uncover more of it by digging in the sand. Again, if not for time constraints, I would look into that furter, as it sounds like a very simple yet enjoyable activity to have built into the AR Sandbox.

One of the more ambitious proposals is to have a 3D model anchored on the sand surface and move around it as it changes. What I expect this to entail is an implementation of ray casting, which would require a bit of work (Kreylos, 2018). From my limited experience in graphics before this project, ray casting, to be done efficiently, requires some degree of identification for each object in the scene, and then identification of every triangle in that object. I've worked with simple triangle meshes, and even for those, the program needs identifying bounding boxes for each object and each triangle in said object. I believe this avenue is possible to succeed in, but will require a very involved level of time and skill to pull off.

To conclude, I deem this AR Sandbox course work a success. I learned a lot about the OpenGL and Vrui toolkits. The latter especially, since unlike OpenGL, Vrui is not widely used or discussed beyond some select projects. This made researching and finding answers to Vrui specific questions a difficult task, as there are very limited resources to pull from. Overall, this project has left me with more knowledge and experience in coding than I had before I started, and I hope my work here contributes to an even more impressive AR Sandbox later down the line.

Bibliography

Brown, W. (2016, March 8). *8.3 - Perspective Projections*¶. LearnWebGL. http://learnwebgl.brown37.net/08_projections/projections_perspective.html.

GLContextData. (2020, February 19). https://web.cs.ucdavis.edu/~okreylos/ResDev/Vrui/Documentation/GLContextData.html.

Hock-Chuan, C. (2012, July). *OpenGL Tutorial, An Introduction on OpenGL with 2D Graphics*. https://www3.ntu.edu.sg/home/ehchua/programming/opengl/cg_introduction.html.

Kreylos, O. (2016). Augmented Reality Sandbox. https://arsandbox.ucdavis.edu/.

Kreylos, O. (2018, June 29). *joystick for local tools, part 2*. Augmented Reality Sandbox. https://arsandbox.ucdavis.edu/forums/topic/joystick-for-local-tools-part-2/.

Kreylos, O. (2020). *Software Installation*. Oliver Kreylos' Research and Development Homepage - Augmented Reality Sandbox. https://web.cs.ucdavis.edu/~okreylos/ResDev/SARndbox/LinkSoftwareInstallation.html.

University of California, Davis. (2016, October 18). *Vrui Documentation and User's Manual for Vrui-4.2-001*. Vrui Documentation and User's Manual. https://web.cs.ucdavis.edu/~okreylos/ResDev/Vrui/Documentation/index.html.

Appendix I: Overview of the AR Sandbox Code

Class/File Name	Description	External Dependencies (Class/File names)
WaterTable2	GLObject that governs how the water simulation moves over a surface	- DepthImageRenderer - ShaderHelper - Types
WaterRenderer	GLObject that renders a water surface. Constructor needs a WaterTable2.	- ShaderHelper - WaterTable2 - Types
Types	Declarations of data types exchanged between the AR Sandbox modules.	- N/A
SurfaceRenderer	GLObject that renders a surface. Requires a DepthImageRenderer in its constructor. This is the class that sets the contour lines in the simulation, dictates some aspects of the water animation, and determines whether or not to illuminate the scene	- DEM - DepthImageRenderer - ElevationColorMap - ShaderHelper - WaterTable2 - Types
ShaderHelper	Helper functions to create GLSL shaders from text files.	- Config
SandboxClient	A GLObject and Vrui application that communicates with a remote AR Sandbox to render its water level and bathymetry.	- N/A
Sandbox	A GLObject and Vrui application to drive an augmented reality sandbox. One of its elements is the remote server that SandboxClient communicates with.	- WaterRenderer - WaterTable2 - SurfaceRenderer - RemoteServer - GlobalWaterTool - LocalWaterTool - HandExtractor - ElevationColorMap - FrameFilter - BathymetrySaverTool - Config - DEM - DEMTool - DepthImageRenderer - Types
RemoteServer	Class to connect bathymetry and water level viewers to an Augmented Reality Sandbox. Its frame() method is called in Sandbox.cpp's frame() method.	- WaterTable2 - Sandbox

LocalWaterTool	Vrui Application Tool class and GLObject to locally add or remove water from an augmented reality sandbox. This adds water at a specific point in the simulation.	- WaterTable2 - Sandbox
HandExtractor	Class to identify hands from a depth image.	- Types
GlobalWaterTool	Vrui Application Tool class to globally add or remove water from an augmented reality sandbox. This is when water is added or removed throughout the simulation	- WaterTable2 - Sandbox
FrameFilter	Class to filter streams of depth frames arriving from a depth camera, with code to detect unstable values in each pixel, and fill holes resulting from invalid samples.	- Types
ElevationColorMap	A GLColorMap and GLTextureObject to represent elevation color maps for topographic maps. Uses a DepthImageRenderer to calculate a texture mapping plane.	- DepthImageRenderer - Types
DepthImageRenderer	A GLObject that communicates with the Kinect in order to render a depth image.	- ShaderHelper - Types
DEMTool	Vrui Application Tool class to load a digital elevation model into an augmented reality sandbox to colorize the sand surface based on distance to the DEM.	- DEM - Types
DEM	Class to represent digital elevation models (DEMs) as float-valued texture objects. To interact with the sandbox topography measurements, this class should be a good starting point.	- Types
Config	Configuration header file for the Augmented Reality Sandbox.	- N/A
CalibrateProjector	Vrui Application Utility to calculate the calibration transformation of a projector into a Kinect-captured 3D space. This will likely be called by anyone attempting to build the AR Sandbox set up and do the necessary calibrations. Unless the project is looking into improving the calibration set up, this is not a class one need concern themselves	- Config
BathymetrySaverTool	Tool to save the current bathymetry grid of an augmented reality sandbox to a file or network socket.	- Sandbox - WaterTable2 - Types

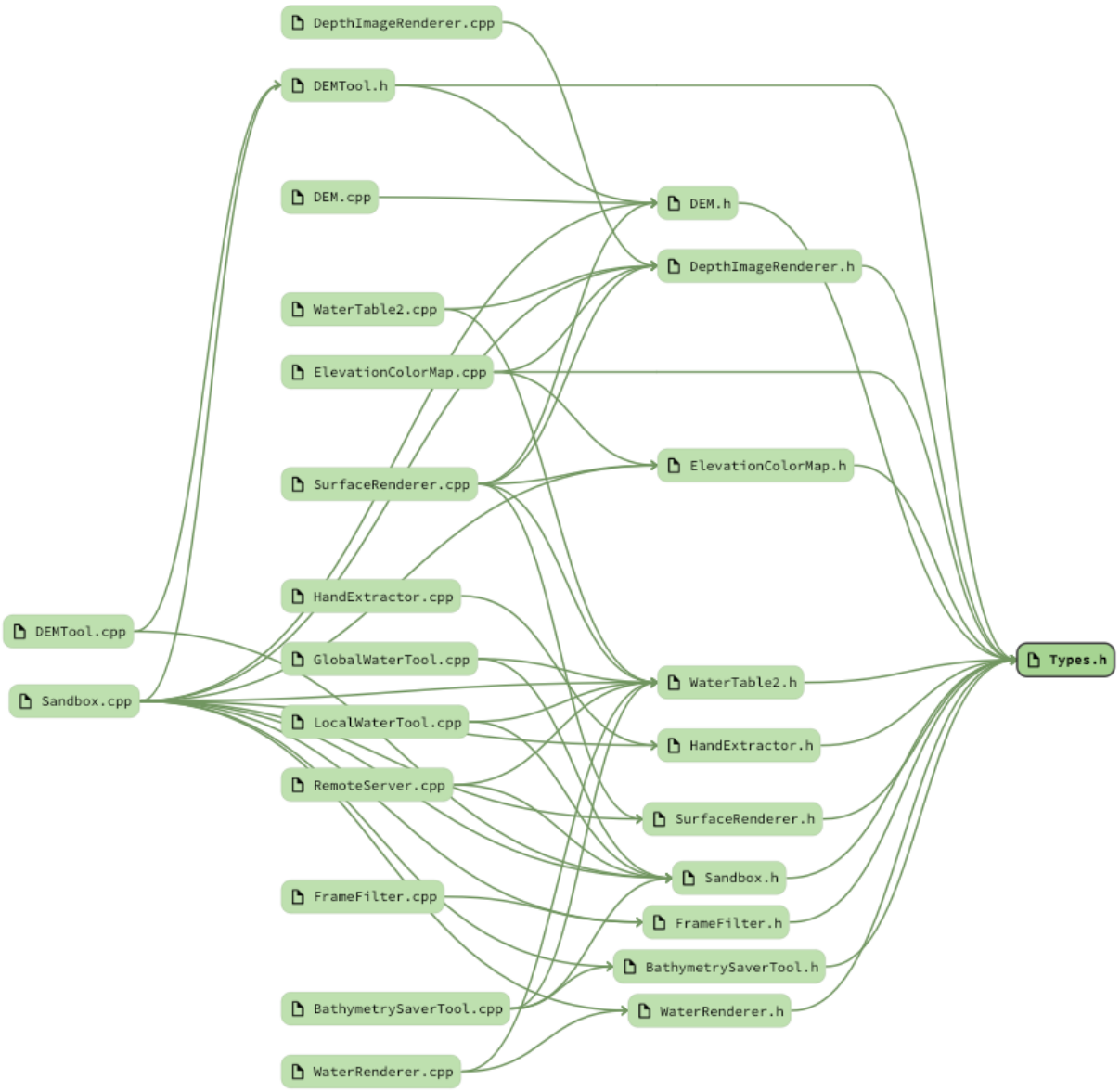

Figure 1 A Sourcetrail map showcasing all the files that depend on Types.h. This is the most all encompasing diagram for the entire AR Sandbox code.

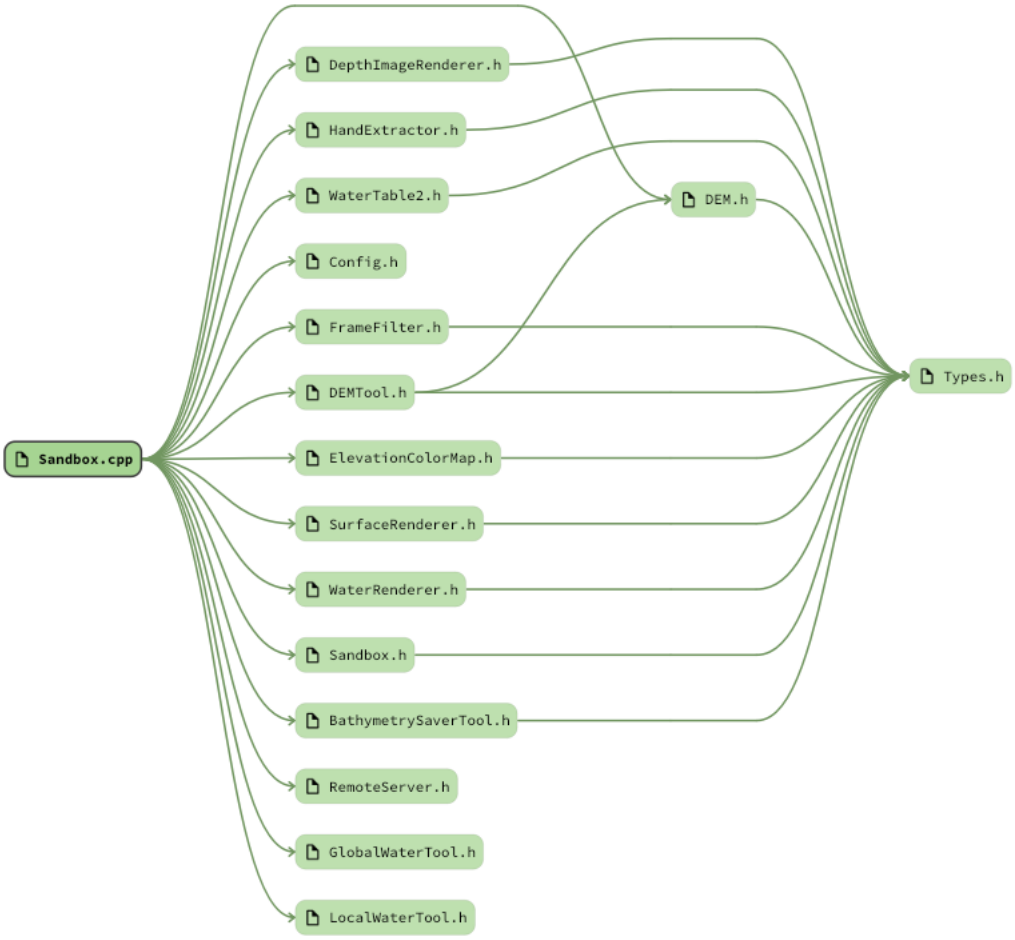

Figure 2 A Sourcetrail Map of all the file dependencies of Sandbox.cpp. Since Sandbox.cpp is the file that drives the AR Sandbox environment, when looking at what to pull from, this is a good map to consider.

Appendix II: Recreating and Compiling the code

The following are copies of the functions that were written and modified for the ball animation. They are separated by file. The code that was written for this project is outlined by yellow asterisks (***).

Sandbox.h

```
class Sandbox:public Vrui::Application,public GLObject
{
    /* Embedded classes: */
private:
    typedef Geometry::Box<Scalar,3> Box; // Type for bounding boxes
    typedef Geometry::OrthonormalTransformation<Scalar,3> ONTransform; // Type for
rigid body transformations
    typedef Kinect::FrameSource::DepthCorrection::PixelCorrection
PixelDepthCorrection; // Type for per-pixel depth correction factors

    struct DataItem:public GLObject::DataItem
    {
        /* Elements: */
    public:
        double waterTableTime; // Simulation time stamp of the water table in this
OpenGL context
        GLsizei shadowBufferSize[2]; // Size of the shadow rendering frame buffer
        GLuint shadowFramebufferObject; // Frame buffer object to render shadow maps
        GLuint shadowDepthTextureObject; // Depth texture for the shadow rendering
frame buffer


        GLuint textureObjectId; // Texture object ID of some texture
        GLuint displayListId; // Display list ID of some display list

        /* Constructors and destructors: */
        DataItem(void);
        virtual ~DataItem(void);
    };

    struct GridRequest // Structure representing a request to read back bathymetry
and/or water level grids from the GPU
    {
        /* Embedded classes: */
    public:
        typedef void (*CallbackFunction)(GLfloat*,GLfloat*,void*); // Type for
callback functions

        struct Request // Structure holding a request's parameters
        {
            /* Elements: */
        public:
            GLfloat* bathymetryBuffer; // Pointer to a buffer to hold the requested
bathymetry grid if requested
            GLfloat* waterLevelBuffer; // Pointer to a buffer to hold the requested
water level grid if requested
            CallbackFunction callback; // Function to call when the grid(s) has/have
```

```
been read back
            void* callbackData; // Additional data element to pass to callback
function

            /* Constructors and destructors: */
            Request(void) // Creates an inactive request

:bathymetryBuffer(0),waterLevelBuffer(0),callback(0),callbackData(0)
            {
            }

            /* Methods: */
            bool isActive(void) const // Returns true if there is a pending request
            {
                return callback!=0;
            }
            void complete(void) // Calls the read-back callback
            {
                (*callback)(bathymetryBuffer,waterLevelBuffer,callbackData);
            }
        };

        /* Elements: */
        Threads::Mutex mutex; // Mutex serializing access to the request structure
        Request currentRequest; // The currently pending grid request

        /* Constructors and destructors: */
        GridRequest(void) // Creates an inactive grid request
        {
        }

        /* Methods: */
        bool requestGrids(GLfloat* newBathymetryBuffer,GLfloat*
newWaterLevelBuffer,CallbackFunction newCallback,void* newCallbackData) // Requests a
grid read-back; returns true if request has been granted
        {
            Threads::Mutex::Lock lock(mutex);
            if(currentRequest.callback==0)
            {
                currentRequest.bathymetryBuffer=newBathymetryBuffer;
                currentRequest.waterLevelBuffer=newWaterLevelBuffer;
                currentRequest.callback=newCallback;
                currentRequest.callbackData=newCallbackData;
                return true;
            }
            else
                return false;
        }
        Request getRequest(void) // Returns the current grid request and deactivates
it
        {
            Threads::Mutex::Lock lock(mutex);
            Request result=currentRequest;
            currentRequest.callback=0;
            return result;
        }
    };

    struct RenderSettings // Structure to hold per-window rendering settings
    {
        /* Elements: */
```

```
    public:
        bool fixProjectorView; // Flag whether to allow viewpoint navigation or always
render from the projector's point of view
        PTransform projectorTransform; // The calibrated projector transformation
matrix for fixed-projection rendering
        bool projectorTransformValid; // Flag whether the projector transformation is
valid
        bool hillshade; // Flag whether to use augmented reality hill shading
        GLMaterial surfaceMaterial; // Material properties to render the surface in
hill shading mode
        bool useShadows; // Flag whether to use shadows in augmented reality hill
shading
        ElevationColorMap* elevationColorMap; // Pointer to an elevation color map
        bool useContourLines; // Flag whether to draw elevation contour lines
        GLfloat contourLineSpacing; // Spacing between adjacent contour lines in cm
        bool renderWaterSurface; // Flag whether to render the water surface as a
geometric surface
        GLfloat waterOpacity; // Opacity factor for water when rendered as texture
        SurfaceRenderer* surfaceRenderer; // Surface rendering object for this window
        WaterRenderer* waterRenderer; // A renderer to render the water surface as
geometry

        /* Constructors and destructors: */
        RenderSettings(void); // Creates default rendering settings
        RenderSettings(const RenderSettings& source); // Copy constructor
        ~RenderSettings(void); // Destroys rendering settings

        /* Methods: */
        void loadProjectorTransform(const char* projectorTransformName); // Loads a
projector transformation from the given file
        void loadHeightMap(const char* heightMapName); // Loads the selected height
map
    };

    friend class GlobalWaterTool;
    friend class LocalWaterTool;
    friend class DEMTool;
    friend class BathymetrySaverTool;
    friend class RemoteServer;

    /* Elements: */

private:
    RemoteServer* remoteServer; // A server to stream bathymetry and water level grids
to remote clients
    Kinect::FrameSource* camera; // The Kinect camera device
    unsigned int frameSize[2]; // Width and height of the camera's depth frames
    PixelDepthCorrection* pixelDepthCorrection; // Buffer of per-pixel depth
correction coefficients
    Kinect::FrameSource::IntrinsicParameters cameraIps; // Intrinsic parameters of the
Kinect camera
    FrameFilter* frameFilter; // Processing object to filter raw depth frames from the
Kinect camera
    bool pauseUpdates; // Pauses updates of the topography
    Threads::TripleBuffer<Kinect::FrameBuffer> filteredFrames; // Triple buffer for
incoming filtered depth frames
    DepthImageRenderer* depthImageRenderer; // Object managing the current filtered
depth image
    ONTransform boxTransform; // Transformation from camera space to baseplane space
(x along long sandbox axis, z up)
    Scalar boxSize; // Radius of sphere around sandbox area
```

```
    Box bbox; // Bounding box around all potential surfaces
    WaterTable2* waterTable; // Water flow simulation object
    double waterSpeed; // Relative speed of water flow simulation
    unsigned int waterMaxSteps; // Maximum number of water simulation steps per frame
    GLfloat rainStrength; // Amount of water deposited by rain tools and objects on
each water simulation step
    HandExtractor* handExtractor; // Object to detect splayed hands above the sand
surface to make rain
    const AddWaterFunction* addWaterFunction; // Render function registered with the
water table
    bool addWaterFunctionRegistered; // Flag if the water adding function is currently
registered with the water table
    mutable GridRequest gridRequest; // Structure holding pending grid read-back
requests
    std::vector<RenderSettings> renderSettings; // List of per-window rendering
settings
    Vrui::Lightsource* sun; // An external fixed light source
    DEM* activeDem; // The currently active DEM
    GLMotif::PopupMenu* mainMenu;
    GLMotif::ToggleButton* pauseUpdatesToggle;
    GLMotif::PopupWindow* waterControlDialog;
    GLMotif::TextFieldSlider* waterSpeedSlider;
    GLMotif::TextFieldSlider* waterMaxStepsSlider;
    GLMotif::TextField* frameRateTextField;
    GLMotif::TextFieldSlider* waterAttenuationSlider;
    int controlPipeFd; // File descriptor of an optional named pipe to send control
commands to a running AR Sandbox



/**************************************************************************************/
    /* Additional code from University of Toronto CSC494 by Philip Smith:
     *
     * These elements work together to control the 'bouncing ball' animation */

    Vrui::Scalar translatedPosition[3]; // The ball's current position in 3D virtual
space
    Vrui::Scalar centerPosition[3]; // The ball's starting position in 3D virtual
space. This is used to determine the boundaries if the viewer moves position
    Vrui::Scalar moveSpeeds[3]; // The ball's current direction of movement in virtual
3D space
    Vrui::Scalar BASEtranslatedPosition[3]; // The ball's current position in 3D
virtual space if centerPosition was (0, 0, 0). This helps when the viewer position
changes
    double ballRadius; // The radius of the ball
    GLfloat ballHMax, ballHMin, ballVMax, ballVMin; // The bounds on which the ball
will bounce off of

/**************************************************************************************/




    /* Private methods: */
    void rawDepthFrameDispatcher(const Kinect::FrameBuffer& frameBuffer); // Callback
receiving raw depth frames from the Kinect camera; forwards them to the frame filter
and rain maker objects
    void receiveFilteredFrame(const Kinect::FrameBuffer& frameBuffer); // Callback
receiving filtered depth frames from the filter object
    void toggleDEM(DEM* dem); // Sets or toggles the currently active DEM
```

```
    void addWater(GLContextData& contextData) const; // Function to render geometry
that adds water to the water table
    void pauseUpdatesCallback(GLMotif::ToggleButton::ValueChangedCallbackData*
cbData);
    void showWaterControlDialogCallback(Misc::CallbackData* cbData);
    void waterSpeedSliderCallback(GLMotif::TextFieldSlider::ValueChangedCallbackData*
cbData);
    void
waterMaxStepsSliderCallback(GLMotif::TextFieldSlider::ValueChangedCallbackData*
cbData);
    void
waterAttenuationSliderCallback(GLMotif::TextFieldSlider::ValueChangedCallbackData*
cbData);
    GLMotif::PopupMenu* createMainMenu(void);
    GLMotif::PopupWindow* createWaterControlDialog(void);

    /* Constructors and destructors: */
public:
    Sandbox(int& argc,char**& argv);
    virtual ~Sandbox(void);

    /* Methods from Vrui::Application: */
    virtual void
toolDestructionCallback(Vrui::ToolManager::ToolDestructionCallbackData* cbData);
    virtual void frame(void);
    virtual void display(GLContextData& contextData) const;
    virtual void resetNavigation(void);
    virtual void eventCallback(EventID eventId,Vrui::InputDevice::ButtonCallbackData*
cbData);

    /* Methods from GLObject: */
    virtual void initContext(GLContextData& contextData) const;




/**********************************************************************************/
    /* Additional code from University of Toronto CSC494 by Philip Smith:
     *
     * This is the helper method(s) used for the 'bouncing ball' animation */

    virtual void findBounds(); // Calculates the boundaries of the ball (ballHMax,
ballHMin, ballVMax, ballVMin)

/**********************************************************************************/
};
```

Sandbox.cpp

```
Sandbox::DataItem::DataItem(void)
        :waterTableTime(0.0),
         shadowFramebufferObject(0),shadowDepthTextureObject(0)
{
    /* Check if all required extensions are supported: */
    bool supported=GLEXTFramebufferObject::isSupported();
    supported=supported&&GLARBTextureRectangle::isSupported();
    supported=supported&&GLARBTextureFloat::isSupported();
    supported=supported&&GLARBTextureRg::isSupported();
    supported=supported&&GLARBDepthTexture::isSupported();
    supported=supported&&GLARBShaderObjects::isSupported();
    supported=supported&&GLARBVertexShader::isSupported();
    supported=supported&&GLARBFragmentShader::isSupported();
    supported=supported&&GLARBMultitexture::isSupported();
    if(!supported)
        Misc::throwStdErr("Sandbox: Not all required extensions are supported by local
OpenGL");

    /* Initialize all required extensions: */
    GLEXTFramebufferObject::initExtension();
    GLARBTextureRectangle::initExtension();
    GLARBTextureFloat::initExtension();
    GLARBTextureRg::initExtension();
    GLARBDepthTexture::initExtension();
    GLARBShaderObjects::initExtension();
    GLARBVertexShader::initExtension();
    GLARBFragmentShader::initExtension();
    GLARBMultitexture::initExtension();


/*****************************************************************************/
    /* Additional code from University of Toronto CSC494 by Philip Smith:
     *
     * These elements are the object Ids that keep track of the data of textures and
GL Objects */

    /* Create a texture object to hold a texture: */
    glGenTextures(1,&textureObjectId);

    /* Create a display list: */
    displayListId=glGenLists(2);

/*****************************************************************************/
}
```

```
Sandbox::DataItem::~DataItem(void)
{
    /* Delete all shaders, buffers, and texture objects: */
    glDeleteFramebuffersEXT(1,&shadowFramebufferObject);
    glDeleteTextures(1,&shadowDepthTextureObject);


/*****************************************************************************/
    /* Additional code from University of Toronto CSC494 by Philip Smith:
     *
     * This code destroys object Ids that were instantiated */

    /* Destroy the texture object: */
    glDeleteTextures(1,&textureObjectId);

    /* Destroy the display list: */
    glDeleteLists(displayListId,1);

/*****************************************************************************/
}
```

```
/*****************************************************************************/
/* Additional code from University of Toronto CSC494 by Philip Smith:
 *
 * Updates centerPosition based on the viewer's position in 3D space. The equation
used is the parametric form of a ray:
 *                 r(t)= <origin vector> + t*<direction vector>
 * Where t is an arbitrary point on the ray. Using this equation I find the center of
the screen according to the latest
 * viewer position. With the updated centerPosition, it finds the horizontal and
vertical bounds based on fixed values*/

void Sandbox::findBounds(){
    centerPosition[0] = Vrui::getHeadPosition()[0] + Vrui::getViewDirection()[0] * 3;
    centerPosition[1] = Vrui::getHeadPosition()[1] + Vrui::getViewDirection()[1] * 3;
    centerPosition[2] = Vrui::getHeadPosition()[2] + Vrui::getViewDirection()[2] * 3;

    ballHMin = centerPosition[0] - 6;
    ballHMax = centerPosition[0] + 5.8;
    ballVMin = centerPosition[1] - 4.4;
    ballVMax = centerPosition[1] + 4.5;
}
/*****************************************************************************/
```

```
Sandbox::Sandbox(int& argc,char**& argv)
        :Vrui::Application(argc,argv),
         remoteServer(0),
         camera(0),pixelDepthCorrection(0),
         frameFilter(0),pauseUpdates(false),
         depthImageRenderer(0),
         waterTable(0),
         handExtractor(0),addWaterFunction(0),addWaterFunctionRegistered(false),
         sun(0),
         activeDem(0),
         mainMenu(0),pauseUpdatesToggle(0),waterControlDialog(0),

waterSpeedSlider(0),waterMaxStepsSlider(0),frameRateTextField(0),waterAttenuationSlide
r(0),
         controlPipeFd(-1)
{
    /* Read the sandbox's default configuration parameters: */
    std::string sandboxConfigFileName=CONFIG_CONFIGDIR;
    sandboxConfigFileName.push_back('/');
    sandboxConfigFileName.append(CONFIG_DEFAULTCONFIGFILENAME);
    Misc::ConfigurationFile sandboxConfigFile(sandboxConfigFileName.c_str());
    Misc::ConfigurationFileSection cfg=sandboxConfigFile.getSection("/SARndbox");
    unsigned int cameraIndex=cfg.retrieveValue<int>("./cameraIndex",0);
    std::string
cameraConfiguration=cfg.retrieveString("./cameraConfiguration","Camera");
    double scale=cfg.retrieveValue<double>("./scaleFactor",100.0);
    std::string sandboxLayoutFileName=CONFIG_CONFIGDIR;
    sandboxLayoutFileName.push_back('/');
    sandboxLayoutFileName.append(CONFIG_DEFAULTBOXLAYOUTFILENAME);

sandboxLayoutFileName=cfg.retrieveString("./sandboxLayoutFileName",sandboxLayoutFileNa
me);
    Math::Interval<double> elevationRange=cfg.retrieveValue<Math::Interval<double>
>("./elevationRange",Math::Interval<double>(-1000.0,1000.0));
    bool haveHeightMapPlane=cfg.hasTag("./heightMapPlane");
    Plane heightMapPlane;
    if(haveHeightMapPlane)
        heightMapPlane=cfg.retrieveValue<Plane>("./heightMapPlane");
    unsigned int numAveragingSlots=cfg.retrieveValue<unsigned
int>("./numAveragingSlots",30);
    unsigned int minNumSamples=cfg.retrieveValue<unsigned int>("./minNumSamples",10);
    unsigned int maxVariance=cfg.retrieveValue<unsigned int>("./maxVariance",2);
    float hysteresis=cfg.retrieveValue<float>("./hysteresis",0.1f);
    Misc::FixedArray<unsigned int,2> wtSize;
    wtSize[0]=640;
    wtSize[1]=480;
    wtSize=cfg.retrieveValue<Misc::FixedArray<unsigned int,2>
>("./waterTableSize",wtSize);
    waterSpeed=cfg.retrieveValue<double>("./waterSpeed",1.0);
    waterMaxSteps=cfg.retrieveValue<unsigned int>("./waterMaxSteps",30U);
    Math::Interval<double> rainElevationRange=cfg.retrieveValue<Math::Interval<double>
>("./rainElevationRange",Math::Interval<double>(-1000.0,1000.0));
    rainStrength=cfg.retrieveValue<GLfloat>("./rainStrength",0.25f);
    double evaporationRate=cfg.retrieveValue<double>("./evaporationRate",0.0);
    float demDistScale=cfg.retrieveValue<float>("./demDistScale",1.0f);
    std::string controlPipeName=cfg.retrieveString("./controlPipeName","");

    /* Process command line parameters: */
    bool printHelp=false;
    const char* frameFilePrefix=0;
```

```
    const char* kinectServerName=0;
    bool useRemoteServer=false;
    int remoteServerPortId=26000;
    int windowIndex=0;
    renderSettings.push_back(RenderSettings());
    for(int i=1;i<argc;++i)
    {
        if(argv[i][0]=='-')
        {
            if(strcasecmp(argv[i]+1,"h")==0)
                printHelp=true;
            else if(strcasecmp(argv[i]+1,"remote")==0)
            {
                /* Check if there is an optional port number: */
                if(i+1<argc&&argv[i+1][0]>='0'&&argv[i+1][0]<='9')
                {
                    ++i;
                    remoteServerPortId=atoi(argv[i]);
                }

                useRemoteServer=true;
            }
            else if(strcasecmp(argv[i]+1,"c")==0)
            {
                ++i;
                cameraIndex=atoi(argv[i]);
            }
            else if(strcasecmp(argv[i]+1,"f")==0)
            {
                ++i;
                frameFilePrefix=argv[i];
            }
            else if(strcasecmp(argv[i]+1,"p")==0)
            {
                ++i;
                kinectServerName=argv[i];
            }
            else if(strcasecmp(argv[i]+1,"s")==0)
            {
                ++i;
                scale=atof(argv[i]);
            }
            else if(strcasecmp(argv[i]+1,"slf")==0)
            {
                ++i;
                sandboxLayoutFileName=argv[i];
            }
            else if(strcasecmp(argv[i]+1,"er")==0)
            {
                ++i;
                double elevationMin=atof(argv[i]);
                ++i;
                double elevationMax=atof(argv[i]);
                elevationRange=Math::Interval<double>(elevationMin,elevationMax);
            }
            else if(strcasecmp(argv[i]+1,"hmp")==0)
            {
                /* Read height mapping plane coefficients: */
                haveHeightMapPlane=true;
                double hmp[4];
                for(int j=0;j<4;++j)
```

```
                {
                    ++i;
                    hmp[j]=atof(argv[i]);
                }
                heightMapPlane=Plane(Plane::Vector(hmp),hmp[3]);
                heightMapPlane.normalize();
            }
            else if(strcasecmp(argv[i]+1,"nas")==0)
            {
                ++i;
                numAveragingSlots=atoi(argv[i]);
            }
            else if(strcasecmp(argv[i]+1,"sp")==0)
            {
                ++i;
                minNumSamples=atoi(argv[i]);
                ++i;
                maxVariance=atoi(argv[i]);
            }
            else if(strcasecmp(argv[i]+1,"he")==0)
            {
                ++i;
                hysteresis=float(atof(argv[i]));
            }
            else if(strcasecmp(argv[i]+1,"wts")==0)
            {
                for(int j=0;j<2;++j)
                {
                    ++i;
                    wtSize[j]=(unsigned int)(atoi(argv[i]));
                }
            }
            else if(strcasecmp(argv[i]+1,"ws")==0)
            {
                ++i;
                waterSpeed=atof(argv[i]);
                ++i;
                waterMaxSteps=atoi(argv[i]);
            }
            else if(strcasecmp(argv[i]+1,"rer")==0)
            {
                ++i;
                double rainElevationMin=atof(argv[i]);
                ++i;
                double rainElevationMax=atof(argv[i]);

rainElevationRange=Math::Interval<double>(rainElevationMin,rainElevationMax);
            }
            else if(strcasecmp(argv[i]+1,"rs")==0)
            {
                ++i;
                rainStrength=GLfloat(atof(argv[i]));
            }
            else if(strcasecmp(argv[i]+1,"evr")==0)
            {
                ++i;
                evaporationRate=atof(argv[i]);
            }
            else if(strcasecmp(argv[i]+1,"dds")==0)
            {
                ++i;
```

```
                demDistScale=float(atof(argv[i]));
            }
            else if(strcasecmp(argv[i]+1,"wi")==0)
            {
                ++i;
                windowIndex=atoi(argv[i]);

                /* Extend the list of render settings if an index beyond the end is
selected: */
                while(int(renderSettings.size())<=windowIndex)
                    renderSettings.push_back(renderSettings.back());

                /* Disable fixed projector view on the new render settings: */
                renderSettings.back().fixProjectorView=false;
            }
            else if(strcasecmp(argv[i]+1,"fpv")==0)
            {
                renderSettings.back().fixProjectorView=true;
                if(i+1<argc&&argv[i+1][0]!='-')
                {
                    /* Load the projector transformation file specified in the next
argument: */
                    ++i;
                    renderSettings.back().loadProjectorTransform(argv[i]);
                }
            }
            else if(strcasecmp(argv[i]+1,"nhs")==0)
                renderSettings.back().hillshade=false;
            else if(strcasecmp(argv[i]+1,"uhs")==0)
                renderSettings.back().hillshade=true;
            else if(strcasecmp(argv[i]+1,"ns")==0)
                renderSettings.back().useShadows=false;
            else if(strcasecmp(argv[i]+1,"us")==0)
                renderSettings.back().useShadows=true;
            else if(strcasecmp(argv[i]+1,"nhm")==0)
            {
                delete renderSettings.back().elevationColorMap;
                renderSettings.back().elevationColorMap=0;
            }
            else if(strcasecmp(argv[i]+1,"uhm")==0)
            {
                if(i+1<argc&&argv[i+1][0]!='-')
                {
                    /* Load the height color map file specified in the next argument:
*/
                    ++i;
                    renderSettings.back().loadHeightMap(argv[i]);
                }
                else
                {
                    /* Load the default height color map: */

renderSettings.back().loadHeightMap(CONFIG_DEFAULTHEIGHTCOLORMAPFILENAME);
                }
            }
            else if(strcasecmp(argv[i]+1,"ncl")==0)
                renderSettings.back().useContourLines=false;
            else if(strcasecmp(argv[i]+1,"ucl")==0)
            {
                renderSettings.back().useContourLines=true;
                if(i+1<argc&&argv[i+1][0]!='-')
```

```
                {
                    /* Read the contour line spacing: */
                    ++i;
                    renderSettings.back().contourLineSpacing=GLfloat(atof(argv[i]));
                }
            }
            else if(strcasecmp(argv[i]+1,"rws")==0)
                renderSettings.back().renderWaterSurface=true;
            else if(strcasecmp(argv[i]+1,"rwt")==0)
                renderSettings.back().renderWaterSurface=false;
            else if(strcasecmp(argv[i]+1,"wo")==0)
            {
                ++i;
                renderSettings.back().waterOpacity=GLfloat(atof(argv[i]));
            }
            else if(strcasecmp(argv[i]+1,"cp")==0)
            {
                ++i;
                controlPipeName=argv[i];
            }
            else
                std::cerr<<"Ignoring unrecognized command line switch
"<<argv[i]<<std::endl;
        }
    }

    /* Print usage help if requested: */
    if(printHelp)
        printUsage();

    if(frameFilePrefix!=0)
    {
        /* Open the selected pre-recorded 3D video files: */
        std::string colorFileName=frameFilePrefix;
        colorFileName.append(".color");
        std::string depthFileName=frameFilePrefix;
        depthFileName.append(".depth");
        camera=new
Kinect::FileFrameSource(IO::openFile(colorFileName.c_str()),IO::openFile(depthFileName
.c_str()));
    }
    else if(kinectServerName!=0)
    {
        /* Split the server name into host name and port: */
        const char* colonPtr=0;
        for(const char* snPtr=kinectServerName;*snPtr!='\0';++snPtr)
            if(*snPtr==':')
                colonPtr=snPtr;
        std::string hostName;
        int port;
        if(colonPtr!=0)
        {
            /* Extract host name and port: */
            hostName=std::string(kinectServerName,colonPtr);
            port=atoi(colonPtr+1);
        }
        else
        {
            /* Use complete host name and default port: */
            hostName=kinectServerName;
            port=26000;
```

```
        }

        /* Open a multiplexed frame source for the given server host name and port
number: */
        Kinect::MultiplexedFrameSource*
source=Kinect::MultiplexedFrameSource::create(Comm::openTCPPipe(hostName.c_str(),port)
);

        /* Use the server's first component stream as the camera device: */
        camera=source->getStream(0);
    }
    else
    {
        /* Open the 3D camera device of the selected index: */
        Kinect::DirectFrameSource*
realCamera=Kinect::openDirectFrameSource(cameraIndex,false);
        Misc::ConfigurationFileSection
cameraConfigurationSection=cfg.getSection(cameraConfiguration.c_str());
        realCamera->configure(cameraConfigurationSection);
        camera=realCamera;
    }
    for(int i=0;i<2;++i)
        frameSize[i]=camera->getActualFrameSize(Kinect::FrameSource::DEPTH)[i];

    /* Get the camera's per-pixel depth correction parameters and evaluate it on the
depth frame's pixel grid: */
    Kinect::FrameSource::DepthCorrection* depthCorrection=camera-
>getDepthCorrectionParameters();
    if(depthCorrection!=0)
    {
        pixelDepthCorrection=depthCorrection->getPixelCorrection(frameSize);
        delete depthCorrection;
    }
    else
    {
        /* Create dummy per-pixel depth correction parameters: */
        pixelDepthCorrection=new PixelDepthCorrection[frameSize[1]*frameSize[0]];
        PixelDepthCorrection* pdcPtr=pixelDepthCorrection;
        for(unsigned int y=0;y<frameSize[1];++y)
            for(unsigned int x=0;x<frameSize[0];++x,++pdcPtr)
            {
                pdcPtr->scale=1.0f;
                pdcPtr->offset=0.0f;
            }
    }

    /* Get the camera's intrinsic parameters: */
    cameraIps=camera->getIntrinsicParameters();

    /* Read the sandbox layout file: */
    Geometry::Plane<double,3> basePlane;
    Geometry::Point<double,3> basePlaneCorners[4];
    {
        IO::ValueSource layoutSource(IO::openFile(sandboxLayoutFileName.c_str()));
        layoutSource.skipWs();

        /* Read the base plane equation: */
        std::string s=layoutSource.readLine();
        basePlane=Misc::ValueCoder<Geometry::Plane<double,3>
>::decode(s.c_str(),s.c_str()+s.length());
        basePlane.normalize();
```

```
        /* Read the corners of the base quadrilateral and project them into the base
plane: */
        for(int i=0;i<4;++i)
        {
            layoutSource.skipWs();
            s=layoutSource.readLine();

basePlaneCorners[i]=basePlane.project(Misc::ValueCoder<Geometry::Point<double,3>
>::decode(s.c_str(),s.c_str()+s.length()));
        }
    }

    /* Limit the valid elevation range to the intersection of the extents of all
height color maps: */
    for(std::vector<RenderSettings>::iterator
rsIt=renderSettings.begin();rsIt!=renderSettings.end();++rsIt)
        if(rsIt->elevationColorMap!=0)
        {
            Math::Interval<double> mapRange(rsIt->elevationColorMap-
>getScalarRangeMin(),rsIt->elevationColorMap->getScalarRangeMax());
            elevationRange.intersectInterval(mapRange);
        }

    /* Scale all sizes by the given scale factor: */
    double sf=scale/100.0; // Scale factor from cm to final units
    for(int i=0;i<3;++i)
        for(int j=0;j<4;++j)
            cameraIps.depthProjection.getMatrix()(i,j)*=sf;

basePlane=Geometry::Plane<double,3>(basePlane.getNormal(),basePlane.getOffset()*sf);
    for(int i=0;i<4;++i)
        for(int j=0;j<3;++j)
            basePlaneCorners[i][j]*=sf;
    if(elevationRange!=Math::Interval<double>::full)
        elevationRange*=sf;
    if(rainElevationRange!=Math::Interval<double>::full)
        rainElevationRange*=sf;
    for(std::vector<RenderSettings>::iterator
rsIt=renderSettings.begin();rsIt!=renderSettings.end();++rsIt)
    {
        if(rsIt->elevationColorMap!=0)
            rsIt->elevationColorMap->setScalarRange(rsIt->elevationColorMap-
>getScalarRangeMin()*sf,rsIt->elevationColorMap->getScalarRangeMax()*sf);
        rsIt->contourLineSpacing*=sf;
        rsIt->waterOpacity/=sf;
        for(int i=0;i<4;++i)
            rsIt->projectorTransform.getMatrix()(i,3)*=sf;
    }
    rainStrength*=sf;
    evaporationRate*=sf;
    demDistScale*=sf;

    /* Create the frame filter object: */
    frameFilter=new
FrameFilter(frameSize,numAveragingSlots,pixelDepthCorrection,cameraIps.depthProjection
,basePlane);
    frameFilter-
>setValidElevationInterval(cameraIps.depthProjection,basePlane,elevationRange.getMin()
,elevationRange.getMax());
    frameFilter->setStableParameters(minNumSamples,maxVariance);
```

```
    frameFilter->setHysteresis(hysteresis);
    frameFilter->setSpatialFilter(true);
    frameFilter-
>setOutputFrameFunction(Misc::createFunctionCall(this,&Sandbox::receiveFilteredFrame))
;

    if(waterSpeed>0.0)
    {
        /* Create the hand extractor object: */
        handExtractor=new
HandExtractor(frameSize,pixelDepthCorrection,cameraIps.depthProjection);
    }

    /* Start streaming depth frames: */
    camera-
>startStreaming(0,Misc::createFunctionCall(this,&Sandbox::rawDepthFrameDispatcher));

    /* Create the depth image renderer: */
    depthImageRenderer=new DepthImageRenderer(frameSize);
    depthImageRenderer->setIntrinsics(cameraIps);
    depthImageRenderer->setBasePlane(basePlane);

    {
        /* Calculate the transformation from camera space to sandbox space: */
        ONTransform::Vector z=basePlane.getNormal();
        ONTransform::Vector x=(basePlaneCorners[1]-
basePlaneCorners[0])+(basePlaneCorners[3]-basePlaneCorners[2]);
        ONTransform::Vector y=z^x;

boxTransform=ONTransform::rotate(Geometry::invert(ONTransform::Rotation::fromBaseVecto
rs(x,y)));
        ONTransform::Point
center=Geometry::mid(Geometry::mid(basePlaneCorners[0],basePlaneCorners[1]),Geometry::
mid(basePlaneCorners[2],basePlaneCorners[3]));
        boxTransform*=ONTransform::translateToOriginFrom(center);

        /* Calculate the size of the sandbox area: */
        boxSize=Geometry::dist(center,basePlaneCorners[0]);
        for(int i=1;i<4;++i)
            boxSize=Math::max(boxSize,Geometry::dist(center,basePlaneCorners[i]));
    }

    /* Calculate a bounding box around all potential surfaces: */
    bbox=Box::empty;
    for(int i=0;i<4;++i)
    {

bbox.addPoint(basePlaneCorners[i]+basePlane.getNormal()*elevationRange.getMin());

bbox.addPoint(basePlaneCorners[i]+basePlane.getNormal()*elevationRange.getMax());
    }

    if(waterSpeed>0.0)
    {
        /* Initialize the water flow simulator: */
        waterTable=new
WaterTable2(wtSize[0],wtSize[1],depthImageRenderer,basePlaneCorners);
        waterTable-
>setElevationRange(elevationRange.getMin(),rainElevationRange.getMax());
        waterTable->setWaterDeposit(evaporationRate);
```

```
        /* Register a render function with the water table: */
        addWaterFunction=Misc::createFunctionCall(this,&Sandbox::addWater);
        waterTable->addRenderFunction(addWaterFunction);
        addWaterFunctionRegistered=true;
    }

    if(useRemoteServer)
    {
        /* Create a remote server: */
        try
        {
            remoteServer=new RemoteServer(this,remoteServerPortId,1.0/30.0);
        }
        catch(const std::runtime_error& err)
        {
            Misc::formattedConsoleError("Sandbox: Unable to create remote server on
port %d due to exception %s",remoteServerPortId,err.what());
        }
    }

    /* Initialize all surface renderers: */
    for(std::vector<RenderSettings>::iterator
rsIt=renderSettings.begin();rsIt!=renderSettings.end();++rsIt)
    {
        /* Calculate the texture mapping plane for this renderer's height map: */
        if(rsIt->elevationColorMap!=0)
        {
            if(haveHeightMapPlane)
                rsIt->elevationColorMap->calcTexturePlane(heightMapPlane);
            else
                rsIt->elevationColorMap->calcTexturePlane(depthImageRenderer);
        }

        /* Initialize the surface renderer: */
        rsIt->surfaceRenderer=new SurfaceRenderer(depthImageRenderer);
        rsIt->surfaceRenderer->setDrawContourLines(rsIt->useContourLines);
        rsIt->surfaceRenderer->setContourLineDistance(rsIt->contourLineSpacing);
        rsIt->surfaceRenderer->setElevationColorMap(rsIt->elevationColorMap);
        rsIt->surfaceRenderer->setIlluminate(rsIt->hillshade);
        if(waterTable!=0)
        {
            if(rsIt->renderWaterSurface)
            {
                /* Create a water renderer: */
                rsIt->waterRenderer=new WaterRenderer(waterTable);
            }
            else
            {
                rsIt->surfaceRenderer->setWaterTable(waterTable);
                rsIt->surfaceRenderer->setAdvectWaterTexture(true);
                rsIt->surfaceRenderer->setWaterOpacity(rsIt->waterOpacity);
            }
        }
        rsIt->surfaceRenderer->setDemDistScale(demDistScale);
    }

#if 0
    /* Create a fixed-position light source: */
   sun=Vrui::getLightsourceManager()->createLightsource(true);
   for(int i=0;i<Vrui::getNumViewers();++i)
      Vrui::getViewer(i)->setHeadlightState(false);
```

```
   sun->enable();
   sun->getLight().position=GLLight::Position(1,0,1,0);
#endif

    /* Create the GUI: */
    mainMenu=createMainMenu();
    Vrui::setMainMenu(mainMenu);
    if(waterTable!=0)
        waterControlDialog=createWaterControlDialog();

    /* Initialize the custom tool classes: */
    GlobalWaterTool::initClass(*Vrui::getToolManager());
    LocalWaterTool::initClass(*Vrui::getToolManager());
    DEMTool::initClass(*Vrui::getToolManager());
    if(waterTable!=0)
        BathymetrySaverTool::initClass(waterTable,*Vrui::getToolManager());
    addEventTool("Pause Topography",0,0);

    if(!controlPipeName.empty())
    {
        /* Open the control pipe in non-blocking mode: */
        controlPipeFd=open(controlPipeName.c_str(),O_RDONLY|O_NONBLOCK);
        if(controlPipeFd<0)
            std::cerr<<"Unable to open control pipe "<<controlPipeName<<";
ignoring"<<std::endl;
    }

    /* Inhibit the screen saver: */
    Vrui::inhibitScreenSaver();

    /* Set the linear unit to support proper scaling: */
    Vrui::getCoordinateManager()-
>setUnit(Geometry::LinearUnit(Geometry::LinearUnit::METER,scale/100.0));



/**********************************************************************************/
    /* Additional code from University of Toronto CSC494 by Philip Smith:
     *
     * These lines are just assigning values to the elements declared in the header
file*/

    moveSpeeds[0]= Vrui::Scalar(0.2);
    moveSpeeds[1]= Vrui::Scalar(0.7);
    moveSpeeds[2]= Vrui::Scalar(0);

    BASEtranslatedPosition[0] = Vrui::Scalar(0);
    BASEtranslatedPosition[1] = Vrui::Scalar(0);
    BASEtranslatedPosition[2] = Vrui::Scalar(0);

    ballRadius = 0.3;

/**********************************************************************************/

}
```

```
void Sandbox::frame(void)
{

    /* Call the remote server's frame method: */
    if(remoteServer!=0)
        remoteServer->frame(Vrui::getApplicationTime());

    /* Check if the filtered frame has been updated: */
    if(filteredFrames.lockNewValue())
    {
        /* Update the depth image renderer's depth image: */
        depthImageRenderer->setDepthImage(filteredFrames.getLockedValue());
    }

    if(handExtractor!=0)
    {
        /* Lock the most recent extracted hand list: */
        handExtractor->lockNewExtractedHands();

#if 0

        /* Register/unregister the rain rendering function based on whether hands have
been detected: */
      bool registerWaterFunction=!handExtractor->getLockedExtractedHands().empty();
      if(addWaterFunctionRegistered!=registerWaterFunction)
         {
         if(registerWaterFunction)
            waterTable->addRenderFunction(addWaterFunction);
         else
            waterTable->removeRenderFunction(addWaterFunction);
         addWaterFunctionRegistered=registerWaterFunction;
         }

#endif
    }

    /* Update all surface renderers: */
    for(std::vector<RenderSettings>::iterator
rsIt=renderSettings.begin();rsIt!=renderSettings.end();++rsIt)
        rsIt->surfaceRenderer->setAnimationTime(Vrui::getApplicationTime());

    /* Check if there is a control command on the control pipe: */
    if(controlPipeFd>=0)
    {
        /* Try reading a chunk of data (will fail with EAGAIN if no data due to non-
blocking access): */
        char commandBuffer[1024];
        ssize_t readResult=read(controlPipeFd,commandBuffer,sizeof(commandBuffer)-1);
        if(readResult>0)
        {
            commandBuffer[readResult]='\0';

            /* Extract commands line-by-line: */
            const char* cPtr=commandBuffer;
            while(*cPtr!='\0')
            {
                /* Split the current line into tokens and skip empty lines: */
                std::vector<std::string> tokens=tokenizeLine(cPtr);
                if(tokens.empty())
                    continue;
```

```
                /* Parse the command: */
                if(isToken(tokens[0],"waterSpeed"))
                {
                    if(tokens.size()==2)
                    {
                        waterSpeed=atof(tokens[1].c_str());
                        if(waterSpeedSlider!=0)
                            waterSpeedSlider->setValue(waterSpeed);
                    }
                    else
                        std::cerr<<"Wrong number of arguments for waterSpeed control
pipe command"<<std::endl;
                }
                else if(isToken(tokens[0],"waterMaxSteps"))
                {
                    if(tokens.size()==2)
                    {
                        waterMaxSteps=atoi(tokens[1].c_str());
                        if(waterMaxStepsSlider!=0)
                            waterMaxStepsSlider->setValue(waterMaxSteps);
                    }
                    else
                        std::cerr<<"Wrong number of arguments for waterMaxSteps
control pipe command"<<std::endl;
                }
                else if(isToken(tokens[0],"waterAttenuation"))
                {
                    if(tokens.size()==2)
                    {
                        double attenuation=atof(tokens[1].c_str());
                        if(waterTable!=0)
                            waterTable->setAttenuation(GLfloat(1.0-attenuation));
                        if(waterAttenuationSlider!=0)
                            waterAttenuationSlider->setValue(attenuation);
                    }
                    else
                        std::cerr<<"Wrong number of arguments for waterAttenuation
control pipe command"<<std::endl;
                }
                else if(isToken(tokens[0],"colorMap"))
                {
                    if(tokens.size()==2)
                    {
                        try
                        {
                            /* Update all height color maps: */
                            for(std::vector<RenderSettings>::iterator
rsIt=renderSettings.begin();rsIt!=renderSettings.end();++rsIt)
                                if(rsIt->elevationColorMap!=0)
                                    rsIt->elevationColorMap->load(tokens[1].c_str());
                        }
                        catch(const std::runtime_error& err)
                        {
                            std::cerr<<"Cannot read height color map "<<tokens[1]<<"
due to exception "<<err.what()<<std::endl;
                        }
                    }
                    else
                        std::cerr<<"Wrong number of arguments for colorMap control
pipe command"<<std::endl;
```

```
                }
                else if(isToken(tokens[0],"heightMapPlane"))
                {
                    if(tokens.size()==5)
                    {
                        /* Read the height map plane equation: */
                        double hmp[4];
                        for(int i=0;i<4;++i)
                            hmp[i]=atof(tokens[1+i].c_str());
                        Plane heightMapPlane=Plane(Plane::Vector(hmp),hmp[3]);
                        heightMapPlane.normalize();

                        /* Override the height mapping planes of all elevation color
maps: */
                        for(std::vector<RenderSettings>::iterator
rsIt=renderSettings.begin();rsIt!=renderSettings.end();++rsIt)
                            if(rsIt->elevationColorMap!=0)
                                rsIt->elevationColorMap-
>calcTexturePlane(heightMapPlane);
                    }
                    else
                        std::cerr<<"Wrong number of arguments for heightMapPlane
control pipe command"<<std::endl;
                }
                else if(isToken(tokens[0],"useContourLines"))
                {
                    if(tokens.size()==2)
                    {
                        /* Parse the command parameter: */
                        if(isToken(tokens[1],"on")||isToken(tokens[1],"off"))
                        {
                            /* Enable or disable contour lines on all surface
renderers: */
                            bool useContourLines=isToken(tokens[1],"on");
                            for(std::vector<RenderSettings>::iterator
rsIt=renderSettings.begin();rsIt!=renderSettings.end();++rsIt)
                                rsIt->surfaceRenderer-
>setDrawContourLines(useContourLines);
                        }
                        else
                            std::cerr<<"Invalid parameter "<<tokens[1]<<" for
useContourLines control pipe command"<<std::endl;
                    }
                    else
                        std::cerr<<"Wrong number of arguments for contourLineSpacing
control pipe command"<<std::endl;
                }
                else if(isToken(tokens[0],"contourLineSpacing"))
                {
                    if(tokens.size()==2)
                    {
                        /* Parse the contour line distance: */
                        GLfloat contourLineSpacing=GLfloat(atof(tokens[1].c_str()));

                        /* Check if the requested spacing is valid: */
                        if(contourLineSpacing>0.0f)
                        {
                            /* Override the contour line spacing of all surface
renderers: */
                            for(std::vector<RenderSettings>::iterator
rsIt=renderSettings.begin();rsIt!=renderSettings.end();++rsIt)
```

```
                                rsIt->surfaceRenderer-
>setContourLineDistance(contourLineSpacing);
                        }
                        else
                            std::cerr<<"Invalid parameter "<<contourLineSpacing<<" for
contourLineSpacing control pipe command"<<std::endl;
                    }
                    else
                        std::cerr<<"Wrong number of arguments for contourLineSpacing
control pipe command"<<std::endl;
                }
                else if(isToken(tokens[0],"dippingBed"))
                {
                    if(tokens.size()==2&&isToken(tokens[1],"off"))
                    {
                        /* Disable dipping bed rendering on all surface renderers: */
                        for(std::vector<RenderSettings>::iterator
rsIt=renderSettings.begin();rsIt!=renderSettings.end();++rsIt)
                            rsIt->surfaceRenderer->setDrawDippingBed(false);
                    }
                    else if(tokens.size()==5)
                    {
                        /* Read the dipping bed plane equation: */
                        GLfloat dbp[4];
                        for(int i=0;i<4;++i)
                            dbp[i]=GLfloat(atof(tokens[1+i].c_str()));
                        SurfaceRenderer::Plane
dippingBedPlane=SurfaceRenderer::Plane(SurfaceRenderer::Plane::Vector(dbp),dbp[3]);
                        dippingBedPlane.normalize();

                        /* Enable dipping bed rendering and set the dipping bed plane
equation on all surface renderers: */
                        for(std::vector<RenderSettings>::iterator
rsIt=renderSettings.begin();rsIt!=renderSettings.end();++rsIt)
                        {
                            rsIt->surfaceRenderer->setDrawDippingBed(true);
                            rsIt->surfaceRenderer-
>setDippingBedPlane(dippingBedPlane);
                        }
                    }
                    else
                        std::cerr<<"Wrong number of arguments for dippingBed control
pipe command"<<std::endl;
                }
                else if(isToken(tokens[0],"foldedDippingBed"))
                {
                    if(tokens.size()==6)
                    {
                        /* Read the dipping bed coefficients: */
                        GLfloat dbc[5];
                        for(int i=0;i<5;++i)
                            dbc[i]=GLfloat(atof(tokens[1+i].c_str()));

                        /* Enable dipping bed rendering and set the dipping bed
coefficients on all surface renderers: */
                        for(std::vector<RenderSettings>::iterator
rsIt=renderSettings.begin();rsIt!=renderSettings.end();++rsIt)
                        {
                            rsIt->surfaceRenderer->setDrawDippingBed(true);
                            rsIt->surfaceRenderer->setDippingBedCoeffs(dbc);
                        }
```

```
                    }
                    else
                        std::cerr<<"Wrong number of arguments for foldedDippingBed
control pipe command"<<std::endl;
                }
                else if(isToken(tokens[0],"dippingBedThickness"))
                {
                    if(tokens.size()==2)
                    {
                        /* Read the dipping bed thickness: */
                        float dippingBedThickness=float(atof(tokens[1].c_str()));

                        /* Set the dipping bed thickness on all surface renderers: */
                        for(std::vector<RenderSettings>::iterator
rsIt=renderSettings.begin();rsIt!=renderSettings.end();++rsIt)
                            rsIt->surfaceRenderer-
>setDippingBedThickness(dippingBedThickness);
                    }
                    else
                        std::cerr<<"Wrong number of arguments for dippingBedThickness
control pipe command"<<std::endl;
                }
                else
                    std::cerr<<"Unrecognized control pipe command
"<<tokens[0]<<std::endl;
            }
        }
    }

    if(frameRateTextField!=0&&Vrui::getWidgetManager()->isVisible(waterControlDialog))
    {
        /* Update the frame rate display: */
        frameRateTextField->setValue(1.0/Vrui::getCurrentFrameTime());
    }

    if(pauseUpdates)
        Vrui::scheduleUpdate(Vrui::getApplicationTime()+1.0/30.0);




/*****************************************************************************/
    /* Additional code from University of Toronto CSC494 by Philip Smith:
     *
     * This updates the state of the 'bouncing ball' animation up to but not including
applying those changes to the model matrix. */

    double frameTime=Vrui::getCurrentFrameTime(); // Get the time that has passed
since the last frame

    Sandbox::findBounds(); // Update the bounds of the animation

    /* If any of the coordinates exceed those bounds, change the ball's direction on
that axis */
    for(int i=0;i<3;++i)
    {

        if (i == 0) {
            if(translatedPosition[i] > ballHMax){
                translatedPosition[i] = ballHMax;
```

```
                moveSpeeds[i] = -moveSpeeds[i];
            }
            else if (translatedPosition[i] < ballHMin) {
                translatedPosition[i] = ballHMin;
                moveSpeeds[i] = -moveSpeeds[i];
            }
        }
        else if (i == 1) {
            if(translatedPosition[i] > ballVMax){
                translatedPosition[i] = ballVMax;
                moveSpeeds[i] = -moveSpeeds[i];
            }
            else if (translatedPosition[i] < ballVMin) {
                translatedPosition[i] = ballVMin;
                moveSpeeds[i] = -moveSpeeds[i];
            }
        }

        /* Calculate the coordinate value that the ball should have moved since the
last frame*/
        BASEtranslatedPosition[i]+= moveSpeeds[i] * frameTime;

        /* Using the viewer position and base translation, find the actual translation
coordinate in the scene*/
        translatedPosition[i] = Vrui::getHeadPosition()[i] + 3 *
Vrui::getViewDirection()[i] + BASEtranslatedPosition[i]; //

    }

    /* Request another rendering cycle to show the animation: */
    Vrui::scheduleUpdate(Vrui::getNextAnimationTime());

/*************************************************************************************/
}
```

```
void Sandbox::display(GLContextData& contextData) const
{
    /* Get the data item: */
    DataItem* dataItem=contextData.retrieveDataItem<DataItem>(this);


/*********************************************************************************/
    /* Additional code from University of Toronto CSC494 by Philip Smith:
     *
     * This applies the translation calculated in frame() to the sphere's model
matrix*/

    glPushMatrix();

    glTranslated(translatedPosition[0], translatedPosition[1], translatedPosition[2]);

    /* Call the display list created in the initDisplay() method: */
    glCallList(dataItem->displayListId);

    glPopMatrix();

/*********************************************************************************/

    /* Get the rendering settings for this window: */
    const Vrui::DisplayState& ds=Vrui::getDisplayState(contextData);
    const Vrui::VRWindow* window=ds.window;
    int windowIndex;

for(windowIndex=0;windowIndex<Vrui::getNumWindows()&&window!=Vrui::getWindow(windowInd
ex);++windowIndex)
        ;
    const RenderSettings&
rs=windowIndex<int(renderSettings.size())?renderSettings[windowIndex]:renderSettings.b
ack();



    /* Check if the water simulation state needs to be updated: */
    if(waterTable!=0&&dataItem->waterTableTime!=Vrui::getApplicationTime())
    {
        /* Retrieve a potential pending grid read-back request: */
        GridRequest::Request request=gridRequest.getRequest();

        /* Update the water table's bathymetry grid: */
        waterTable->updateBathymetry(contextData);

        /* Check if the grid request is active and wants bathymetry data: */
        if(request.isActive()&&request.bathymetryBuffer!=0)
        {
            /* Read back the current bathymetry grid: */
            waterTable->bindBathymetryTexture(contextData);

glGetTexImage(GL_TEXTURE_RECTANGLE_ARB,0,GL_RED,GL_FLOAT,request.bathymetryBuffer);
            glBindTexture(GL_TEXTURE_RECTANGLE_ARB,0);
        }

        /* Run the water flow simulation's main pass: */
        GLfloat totalTimeStep=GLfloat(Vrui::getFrameTime()*waterSpeed);
        unsigned int numSteps=0;
        while(numSteps<waterMaxSteps-1U&&totalTimeStep>1.0e-8f)
```

```
        {
            /* Run with a self-determined time step to maintain stability: */
            waterTable->setMaxStepSize(totalTimeStep);
            GLfloat timeStep=waterTable->runSimulationStep(false,contextData);
            totalTimeStep-=timeStep;
            ++numSteps;
        }
#if 0
        if(totalTimeStep>1.0e-8f)
         {
         std::cout<<'.'<<std::flush;
         /* Force the final step to avoid simulation slow-down: */
         waterTable->setMaxStepSize(totalTimeStep);
         GLfloat timeStep=waterTable->runSimulationStep(true,contextData);
         totalTimeStep-=timeStep;
         ++numSteps;
         }
#else
        if(totalTimeStep>1.0e-8f)
            std::cout<<"Ran out of time by "<<totalTimeStep<<std::endl;
#endif

        /* Check if the grid request is active and wants water level data: */
        if(request.isActive()&&request.waterLevelBuffer!=0)
        {
            /* Read back the current water level grid: */
            waterTable->bindQuantityTexture(contextData);

glGetTexImage(GL_TEXTURE_RECTANGLE_ARB,0,GL_RED,GL_FLOAT,request.waterLevelBuffer);
            glBindTexture(GL_TEXTURE_RECTANGLE_ARB,0);
        }

        /* Finish an active grid request: */
        if(request.isActive())
            request.complete();

        /* Mark the water simulation state as up-to-date for this frame: */
        dataItem->waterTableTime=Vrui::getApplicationTime();
    }

    /* Calculate the projection matrix: */
    PTransform projection=ds.projection;
    if(rs.fixProjectorView&&rs.projectorTransformValid)
    {
        /* Use the projector transformation instead: */
        projection=rs.projectorTransform;

        /* Multiply with the inverse modelview transformation so that lighting still
works as usual: */
        projection*=Geometry::invert(ds.modelviewNavigational);
    }

    if(rs.hillshade)
    {
        /* Set the surface material: */
        glMaterial(GLMaterialEnums::FRONT,rs.surfaceMaterial);
    }

#if 0
    if(rs.hillshade&&rs.useShadows)
      {
```

```
      /* Set up OpenGL state: */

glPushAttrib(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT|GL_ENABLE_BIT|GL_POLYGON_BIT);

      GLLightTracker& lt=*contextData.getLightTracker();

      /* Save the currently-bound frame buffer and viewport: */
      GLint currentFrameBuffer;
      glGetIntegerv(GL_FRAMEBUFFER_BINDING_EXT,&currentFrameBuffer);
      GLint currentViewport[4];
      glGetIntegerv(GL_VIEWPORT,currentViewport);

      /**********************************************************************
      First rendering pass: Global ambient illumination only
      **********************************************************************/

      /* Draw the surface mesh: */
      surfaceRenderer->glRenderGlobalAmbientHeightMap(dataItem-
>heightColorMapObject,contextData);

      /**********************************************************************
      Second rendering pass: Add local illumination for every light source
      **********************************************************************/

      /* Enable additive rendering: */
      glEnable(GL_BLEND);
      glBlendFunc(GL_ONE,GL_ONE);
      glDepthFunc(GL_LEQUAL);
      glDepthMask(GL_FALSE);

      for(int
lightSourceIndex=0;lightSourceIndex<lt.getMaxNumLights();++lightSourceIndex)
         if(lt.getLightState(lightSourceIndex).isEnabled())
            {
            /*****************************************************************
            First step: Render to the light source's shadow map
            *****************************************************************/

            /* Set up OpenGL state to render to the shadow map: */
            glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,dataItem-
>shadowFramebufferObject);
            glViewport(0,0,dataItem->shadowBufferSize[0],dataItem-
>shadowBufferSize[1]);
            glDepthMask(GL_TRUE);
            glClear(GL_DEPTH_BUFFER_BIT);
            glCullFace(GL_FRONT);

            /*****************************************************************
            Calculate the shadow projection matrix:
            *****************************************************************/

            /* Get the light source position in eye space: */
            Geometry::HVector<float,3> lightPosEc;

glGetLightfv(GL_LIGHT0+lightSourceIndex,GL_POSITION,lightPosEc.getComponents());

            /* Transform the light source position to camera space: */
            Vrui::ONTransform::HVector
lightPosCc=Vrui::getDisplayState(contextData).modelviewNavigational.inverseTransform(V
rui::ONTransform::HVector(lightPosEc));
```

```
            /* Calculate the direction vector from the center of the bounding box to
the light source: */
            Point bboxCenter=Geometry::mid(bbox.min,bbox.max);
            Vrui::Vector lightDirCc=Vrui::Vector(lightPosCc.getComponents())-
Vrui::Vector(bboxCenter.getComponents())*lightPosCc[3];

            /* Build a transformation that aligns the light direction with the
positive z axis: */
            Vrui::ONTransform
shadowModelview=Vrui::ONTransform::rotate(Vrui::Rotation::rotateFromTo(lightDirCc,Vrui
::Vector(0,0,1)));
            shadowModelview*=Vrui::ONTransform::translateToOriginFrom(bboxCenter);

            /* Create a projection matrix, based on whether the light is positional or
directional: */
            PTransform shadowProjection(0.0);
            if(lightPosEc[3]!=0.0f)
               {
               /* Modify the modelview transformation such that the light source is at
the origin: */

shadowModelview.leftMultiply(Vrui::ONTransform::translate(Vrui::Vector(0,0,-
lightDirCc.mag())));

               /************************************************************
               Create a perspective projection:
               ************************************************************/

               /* Calculate the perspective bounding box of the surface bounding box
in eye space: */
               Box pBox=Box::empty;
               for(int i=0;i<8;++i)
                  {
                  Point bc=shadowModelview.transform(bbox.getVertex(i));
                  pBox.addPoint(Point(-bc[0]/bc[2],-bc[1]/bc[2],-bc[2]));
                  }

               /* Upload the frustum matrix: */
               double l=pBox.min[0]*pBox.min[2];
               double r=pBox.max[0]*pBox.min[2];
               double b=pBox.min[1]*pBox.min[2];
               double t=pBox.max[1]*pBox.min[2];
               double n=pBox.min[2];
               double f=pBox.max[2];
               shadowProjection.getMatrix()(0,0)=2.0*n/(r-l);
               shadowProjection.getMatrix()(0,2)=(r+l)/(r-l);
               shadowProjection.getMatrix()(1,1)=2.0*n/(t-b);
               shadowProjection.getMatrix()(1,2)=(t+b)/(t-b);
               shadowProjection.getMatrix()(2,2)=-(f+n)/(f-n);
               shadowProjection.getMatrix()(2,3)=-2.0*f*n/(f-n);
               shadowProjection.getMatrix()(3,2)=-1.0;
               }
            else
               {
               /************************************************************
               Create a perspective projection:
               ************************************************************/

               /* Transform the bounding box with the modelview transformation: */
               Box bboxEc=bbox;
               bboxEc.transform(shadowModelview);
```

```
               /* Upload the ortho matrix: */
               double l=bboxEc.min[0];
               double r=bboxEc.max[0];
               double b=bboxEc.min[1];
               double t=bboxEc.max[1];
               double n=-bboxEc.max[2];
               double f=-bboxEc.min[2];
               shadowProjection.getMatrix()(0,0)=2.0/(r-l);
               shadowProjection.getMatrix()(0,3)=-(r+l)/(r-l);
               shadowProjection.getMatrix()(1,1)=2.0/(t-b);
               shadowProjection.getMatrix()(1,3)=-(t+b)/(t-b);
               shadowProjection.getMatrix()(2,2)=-2.0/(f-n);
               shadowProjection.getMatrix()(2,3)=-(f+n)/(f-n);
               shadowProjection.getMatrix()(3,3)=1.0;
               }

            /* Multiply the shadow modelview matrix onto the shadow projection matrix:
*/
            shadowProjection*=shadowModelview;

            /* Draw the surface into the shadow buffer: */
            surfaceRenderer->glRenderDepthOnly(shadowProjection,contextData);

            /* Reset OpenGL state: */
            glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,currentFrameBuffer);

glViewport(currentViewport[0],currentViewport[1],currentViewport[2],currentViewport[3]
);
            glCullFace(GL_BACK);
            glDepthMask(GL_FALSE);

            #if SAVEDEPTH
            /* Save the depth image: */
            {
            glBindTexture(GL_TEXTURE_2D,dataItem->shadowDepthTextureObject);
            GLfloat* depthTextureImage=new GLfloat[dataItem-
>shadowBufferSize[1]*dataItem->shadowBufferSize[0]];

glGetTexImage(GL_TEXTURE_2D,0,GL_DEPTH_COMPONENT,GL_FLOAT,depthTextureImage);
            glBindTexture(GL_TEXTURE_2D,0);
            Images::RGBImage dti(dataItem->shadowBufferSize[0],dataItem-
>shadowBufferSize[1]);
            GLfloat* dtiPtr=depthTextureImage;
            Images::RGBImage::Color* ciPtr=dti.modifyPixels();
            for(int y=0;y<dataItem->shadowBufferSize[1];++y)
               for(int x=0;x<dataItem->shadowBufferSize[0];++x,++dtiPtr,++ciPtr)
                  {
                  GLColor<GLfloat,3> tc(*dtiPtr,*dtiPtr,*dtiPtr);
                  *ciPtr=tc;
                  }
            delete[] depthTextureImage;
            Images::writeImageFile(dti,"DepthImage.png");
            }
            #endif

            /* Draw the surface using the shadow texture: */
            rs.surfaceRenderer->glRenderShadowedIlluminatedHeightMap(dataItem-
>heightColorMapObject,dataItem-
>shadowDepthTextureObject,shadowProjection,contextData);
            }
```

```
      /* Reset OpenGL state: */
      glPopAttrib();
      }
   else
#endif
    {
        /* Render the surface in a single pass: */
        rs.surfaceRenderer-
>renderSinglePass(ds.viewport,projection,ds.modelviewNavigational,contextData);
    }

    if(rs.waterRenderer!=0)
    {
        /* Draw the water surface: */

glMaterialAmbientAndDiffuse(GLMaterialEnums::FRONT,GLColor<GLfloat,4>(0.0f,0.5f,0.8f))
;
        glMaterialSpecular(GLMaterialEnums::FRONT,GLColor<GLfloat,4>(1.0f,1.0f,1.0f));
        glMaterialShininess(GLMaterialEnums::FRONT,64.0f);
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
        rs.waterRenderer->render(projection,ds.modelviewNavigational,contextData);
        glDisable(GL_BLEND);
    }

    /* Call the remote server's render method: */
    if(remoteServer!=0)
    {
        glMatrixMode(GL_PROJECTION);
        glPushMatrix();
        glLoadMatrix(projection);
        glMatrixMode(GL_MODELVIEW);
        remoteServer->glRenderAction(contextData);
        glMatrixMode(GL_PROJECTION);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
    }

}
```

```
void Sandbox::resetNavigation(void)
{
    /* Construct a navigation transformation to center the sandbox area in the
display, facing the viewer, with the long sandbox axis facing to the right: */
    Vrui::NavTransform
nav=Vrui::NavTransform::translateFromOriginTo(Vrui::getDisplayCenter());
    nav*=Vrui::NavTransform::scale(Vrui::getDisplaySize()/boxSize);
    Vrui::Vector y=Vrui::getUpDirection();
    Vrui::Vector z=Vrui::getForwardDirection();
    Vrui::Vector x=z^y;
    nav*=Vrui::NavTransform::rotate(Vrui::Rotation::fromBaseVectors(x,y));

    nav*=boxTransform;
    Vrui::setNavigationTransformation(nav);



/*****************************************************************************/
    /* Additional code from University of Toronto CSC494 by Philip Smith:
    *
    * This code starts the ball off in the center of the screen and in front of the
depth map no matter the initial navigation transformation. It uses the logic of the
parametric ray equation:
    *             r(t) = <origin vector> + <direction vector> * t
    * where t is an arbitrary point on the line*/

    translatedPosition[0] = Vrui::getHeadPosition()[0] + Vrui::getViewDirection()[0] *
3;
    translatedPosition[1] = Vrui::getHeadPosition()[1] + Vrui::getViewDirection()[1] *
3;
    translatedPosition[2] = Vrui::getHeadPosition()[2] + Vrui::getViewDirection()[2] *
3;

/*****************************************************************************/
}
```

```
void Sandbox::initContext(GLContextData& contextData) const
{
    /* Create a data item and add it to the context: */
    DataItem* dataItem=new DataItem;
    contextData.addDataItem(this,dataItem);


    {


/*****************************************************************************/
        /* Additional code from University of Toronto CSC494 by Philip Smith:
        *
        * This initializes the 'ball' in the animation. */

        /* Upload all display lists' contents: */
        glNewList(dataItem->displayListId,GL_COMPILE);

        /* Draws the sphere that will be the 'ball' in the animation */
        glDrawSphereIcosahedron(ballRadius,4);

        /* Finish the display list: */
        glEndList();

/*****************************************************************************/


        /* Save the currently bound frame buffer: */
        GLint currentFrameBuffer;
        glGetIntegerv(GL_FRAMEBUFFER_BINDING_EXT,&currentFrameBuffer);

        /* Set the default shadow buffer size: */
        dataItem->shadowBufferSize[0]=1024;
        dataItem->shadowBufferSize[1]=1024;

        /* Generate the shadow rendering frame buffer: */
        glGenFramebuffersEXT(1,&dataItem->shadowFramebufferObject);
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,dataItem->shadowFramebufferObject);

        /* Generate a depth texture for shadow rendering: */
        glGenTextures(1,&dataItem->shadowDepthTextureObject);
        glBindTexture(GL_TEXTURE_2D,dataItem->shadowDepthTextureObject);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);

glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_COMPARE_MODE_ARB,GL_COMPARE_R_TO_TEXTURE);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_COMPARE_FUNC_ARB,GL_LEQUAL);
        glTexParameteri(GL_TEXTURE_2D,GL_DEPTH_TEXTURE_MODE_ARB,GL_INTENSITY);
        glTexImage2D(GL_TEXTURE_2D,0,GL_DEPTH_COMPONENT24_ARB,dataItem-
>shadowBufferSize[0],dataItem-
>shadowBufferSize[1],0,GL_DEPTH_COMPONENT,GL_UNSIGNED_BYTE,0);
        glBindTexture(GL_TEXTURE_2D,0);

        /* Attach the depth texture to the frame buffer object: */

glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,GL_DEPTH_ATTACHMENT_EXT,GL_TEXTURE_2D,dat
aItem->shadowDepthTextureObject,0);
        glDrawBuffer(GL_NONE);
```

```
        glReadBuffer(GL_NONE);
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,currentFrameBuffer);

    }
```

VruiDemoSmall.cpp

```
/***********************************************************************
VruiDemoSmall - Extremely simple Vrui application to demonstrate the
small amount of code overhead introduced by the Vrui toolkit.
Copyright (c) 2006-2015 Oliver Kreylos

This program is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 2 of the License, or (at your
option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
***********************************************************************/

#include <Math/Math.h>
#include <Geometry/OrthogonalTransformation.h>
#include <GL/gl.h>
#include <GL/GLColorTemplates.h>
#include <GL/GLMaterialTemplates.h>
#include <GL/GLVertexTemplates.h>
#include <GL/GLObject.h>
#include <GL/GLContextData.h>
#include <GL/GLGeometryWrappers.h>
#include <GL/GLModels.h>
#include <GL/GLWindow.h>
#include <Vrui/Vrui.h>
#include <Vrui/Application.h>
#include <Vrui/VRScreen.h>
#include <Vrui/VRWindow.h>
#include <random>
#include <iostream>

class VruiDemoSmall:public Vrui::Application, public GLObject
   {
    /* Embedded classes: */
   private:
   struct DataItem:public GLObject::DataItem // Data structure storing OpenGL-
dependent application data
      {
      /* Elements: */
      public:
      GLuint textureObjectId; // Texture object ID of some texture
      GLuint displayListId; // Display list ID of some display list

      /* Constructors and destructors: */
      DataItem(void)
         {
```

```
        /* Create a texture object to hold a texture: */
        glGenTextures(1,&textureObjectId);

        /* Create a display list: */
        displayListId=glGenLists(1);
        };
      virtual ~DataItem(void)
        {
        /* Destroy the texture object: */
        glDeleteTextures(1,&textureObjectId);

        /* Destroy the display list: */
        glDeleteLists(displayListId,1);
        };
      };

    /* Elements: */
   private:
/*****************************************************************************/
/*
/*  Vrui::Scalar transPos[3];
/*    Vrui::Scalar moveSpeeds[3];
/*    Vrui::Scalar masterSpeeds[3];
/*    double ballRadius;
/*    GLfloat ballXMax, ballXMin, ballZMax, ballZMin;
/*****************************************************************************/









   /* Constructors and destructors: */
   public:
   VruiDemoSmall(int& argc,char**& argv);
    virtual ~VruiDemoSmall(void); // Shuts down the Vrui toolkit

   /* Methods from Vrui::Application: */
   virtual void frame(void); // Called exactly once per frame
   virtual void display(GLContextData& contextData) const;
   virtual void resetNavigation(void);

    /* Methods from GLObject: */
   virtual void initContext(GLContextData& contextData) const; // Called once upon
creation of each OpenGL context









/*****************************************************************************/
/*
/*   /* Method for ball animation */
/*    virtual void randomDistribution(void);
/*    virtual void findShape(int width, int height);
/*****************************************************************************/
```

```
};




/*****************************
Methods of class VruiDemoSmall:
*****************************/

VruiDemoSmall::VruiDemoSmall(int& argc,char**& argv)
   :Vrui::Application(argc,argv)
   {
/*************************************************************************************/
/*
/*    /* Initialize the animation parameters: */
/*   for(int i=0;i<3;++i)
/*      transPos[i]=Vrui::Scalar(0);
/*   masterSpeeds[0] = Vrui::Scalar(0.05);
/*   masterSpeeds[1] = Vrui::Scalar(0);
/*   masterSpeeds[2] = Vrui::Scalar(0.1);
/*   moveSpeeds[0]= masterSpeeds[0];
/*   moveSpeeds[1]= masterSpeeds[1];
/*   moveSpeeds[2]= masterSpeeds[2];
/*   ballRadius = 0.5;
/*
/*************************************************************************************/

   }







/*************************************************************************************/
/*
/*void VruiDemoSmall::findShape(int width, int height){
/*
/*    ballXMin = -0.68;
/*    ballXMax = 0.68;
/*    ballZMin = -0.68;
/*    ballZMax = 0.68;
/*
/*}
/*************************************************************************************/




VruiDemoSmall::~VruiDemoSmall(void)
   {
```

```
   }




/*****************************************************************************/
/*
/*void VruiDemoSmall::randomDistribution(void)
/*{
/*    std::random_device rd;
/*    std::uniform_real_distribution<> xDis((masterSpeeds[0] - 2.5), (masterSpeeds[0]
/*+ 2.5));
/*    std::uniform_real_distribution<> zDis((masterSpeeds[2] - 2.5), (masterSpeeds[2]
/*+ 2.5));
/*    std::mt19937 gen(rd());
/*    moveSpeeds[0] = xDis(gen);
/*    moveSpeeds[2] = zDis(gen);
/*
/*}
/*****************************************************************************/









void VruiDemoSmall::frame(void)
   {
   /*************************************************************************
   This function is called exactly once per frame, no matter how many
   eyes or windows exist. It is the perfect place to change application
   or Vrui state (run simulations, animate models, synchronize with
   background threads, change the navigation transformation, etc.).
   *************************************************************************/

/*****************************************************************************/
/*
/*   /* Get the time since the last frame: */
/*   double frameTime=Vrui::getCurrentFrameTime();
/*
/*
/*    Vrui::VRWindow* thisWindow = Vrui::getWindow(0);
/*    int windowSize[2] = {thisWindow->getViewportSize()[0], thisWindow-
/*>getViewportSize()[1]};
/*    VruiDemoSmall::findShape(windowSize[0], windowSize[1]);
/*   /* Change the model angles: */
/*   for(int i=0;i<3;++i)
/*      {
/*        if (i == 0 && transPos[i] > ballXMax) {
/*            transPos[i] = ballXMax;
/*            moveSpeeds[i] = -moveSpeeds[i];
/*            masterSpeeds[i] = -masterSpeeds[i];
```

```
/*
/*        } else if (i==0 && transPos[i] < ballXMin) {
/*            transPos[i] = ballXMin;
/*            moveSpeeds[i] = -moveSpeeds[i];
/*            masterSpeeds[i] = -masterSpeeds[i];
/*
/*        } else if (i == 2 && transPos[i] > ballZMax) {
/*            transPos[i] = ballZMax;
/*            moveSpeeds[i] = -moveSpeeds[i];
/*            masterSpeeds[i] = -masterSpeeds[i];
/*
/*        } else if (i==2 && transPos[i] < ballZMin) {
/*            transPos[i] = ballZMin;
/*            moveSpeeds[i] = -moveSpeeds[i];
/*            masterSpeeds[i] = -masterSpeeds[i];
/*        }
/*
/*
/*      transPos[i]+=moveSpeeds[i]*frameTime;
/*      }
/*
/* /* Request another rendering cycle to show the animation: */
/*
/*   Vrui::scheduleUpdate(Vrui::getNextAnimationTime());
/*****************************************************************************/

   }



void VruiDemoSmall::display(GLContextData& contextData) const
   {
    /* Get the OpenGL-dependent application data from the GLContextData object: */
   DataItem* dataItem=contextData.retrieveDataItem<DataItem>(this);

/*****************************************************************************/
/*
/*
/*    /* Save OpenGL state: */
/*    glPushAttrib(GL_TRANSFORM_BIT);
/*
/*
/*    glPushMatrix();
/*
/*    glTranslated(transPos[0], transPos[1], transPos[2]);
/*
/*    /* Call the display list created in the initDisplay() method: */
/*    glCallList(dataItem->displayListId);
/*
/*
/*    glPopMatrix();
/*
/*
/*    glPushMatrix();
/*
/*    glTranslated(-5.0,0.0,0.0);
/*
/*glMaterialAmbientAndDiffuse(GLMaterialEnums::FRONT,GLColor<GLfloat,4>(1.0f,0.5f,0.5f
/*));
/*    glDrawCube(2.5f);
/*
```

```
/*    glPopMatrix();
/*
/*    /* Restore OpenGL state: */
/*   glPopAttrib();
/*****************************************************************************/

   }


void VruiDemoSmall::resetNavigation(void)
   {
   /* Set the navigation transformation to show the entire scene: */
   Vrui::setNavigationTransformation(Vrui::Point::origin,Vrui::Scalar(1));
   }

void VruiDemoSmall::initContext(GLContextData& contextData) const
   {
   /*********************************************************************
   For classes derived from GLObject, this method is called for each
   newly-created object, once per every OpenGL context created by Vrui.
   This method must not change application or Vrui state, but only create
   OpenGL-dependent application data and store them in the GLContextData
   object for retrieval in the display method.
   *********************************************************************/

   /* Create context data item and store it in the GLContextData object: */
   DataItem* dataItem=new DataItem;
   contextData.addDataItem(this,dataItem);


/*****************************************************************************/
/*
/*   /* Now is also the time to upload all display lists' contents: */
/*   glNewList(dataItem->displayListId,GL_COMPILE);
/*
/*
/*    glDrawSphereIcosahedron(0.02,4);
/*
/*
/*    /* Finish the display list: */
/*   glEndList();
/*****************************************************************************/

   }

/* Create and execute an application object: */
VRUI_APPLICATION_RUN(VruiDemoSmall)
```

Assuming the indicated sections have been copied and pasted into the indicated files and functions, one can just compile the AR Sandbox code and run the program with the ball animation. This can be done by running in the command line:

```
cd ~/src/SARndbox-2.7
make
./bin/SARndbox -uhm -fpv
```

And since this was tested without the water simulation, in case there are problems with that feature, replace the last line with:

```
./bin/SARndbox -uhm -fpv -ws 0 0.0
```

Do note that the location of the files may vary if the AR Sandbox folders were not built as specified in the instructions (Kreylos, 2020). Additionally, the AR Sandbox might come out with a later version than what's specified there. So the first line is subject to change depending on that.

Also, since the panning viewport feature of Vrui is defaulted to be active, this is the line for accessing the file to change that:

```
sudo xed /usr/local/etc/Vrui-5.2/Vrui.cfg
```

`VruiDemoSmall.cpp` is a prototype of the final solution. There are some features and fine tuning that was done in the AR Sandbox code that didn't make it back to the prototype. However, should someone want to play around with an almost empty Vrui space with some animation code already included, these are the command lines to compile and run:

```
cd ~/src/Vrui-5.2/ExamplePrograms
make
./bin/VruiDemoSmall
```

Again, the first line is subject to change due to the reasons explained before.