# Books of Sand Documentation

# <u>Books of Sand Project Overview</u>

**Project goals**

- Want users to be able to interact with words through the medium of sand
    - Use geosciences SARndbox open source program as foundation
        - A program written by a UC Davis professor that projects topographical color maps onto sand and updates in real time as sand is moved around
        - Instead of different colors at different depths, we want to use text of different pages at different depths
- Height / depth of sand determines what text is displayed
    - Making a gap is like tearing a hole through to another page
- Input text files for display
- Screenshot capabilities

**Project info**

- Original researchers: Hamilton College Computer Science 2018 seniors
    - Eseosa Asiruwa
    - Lia Jundt
    - Maya Montgomery
- Main contact
    - Andrew Rippeon, Visiting Assistant Professor of Literature
        - Head of project
        - Built the physical sandbox
- Location
    - Top floor of Root academic hall in a locked attic room

**Project status**

- <u>Complete</u>
- Great deal of research examining the SARnbox project code
- Hardware and software set up and calibrated
- Graphical component of depth-dependent text display

- <u>Future work</u>
- Finish connecting XBox Kinect to graphical application for real-time depth maps using existing Kinect/Vrui code

**Useful resources**

- Books of Sand Github (contains all code for this project)
- UC Davis Official SARndbox Site
- UC Davis SARndbox Forum
- UC Davis SARndbox Tech FAQs
- Steve Young's Installation Script & Guide
- Vrui UI Github
- Kinect Github
- SARndbox Github
- Augmented Reality Sandbox with Real-Time Water Flow Simulation (Video)

**All involved parties**

- Andrew Rippeon (Lit. & Creative Writing) - Head of project, provided goals and management, built box.
- Steve Young (HPC) - Provided software support and CentOS installation.
- Dave Tewksbury (GEOSC) - Provided access to the Geosciences department sandbox (running SARndbox) and advice.
- Rick Decker (CPSCI) - Provided project management for original researchers.
- Jerry Tylutki (CPSCI) - Assisted with security considerations, software installation.
- Kyle Burnham, Ben Salzman (R&ID) - Academic/digital media support, provided XBox Kinect.
- Janet Simons, Greg Lord (DHi) - Assisted with project management.
- Adam Wickert (DIS) - Provided computer hardware.
- Tim Hicks (A/V) - Provided support for the projector.
- John Powell (Studio Art) - Assisted with construction and design.
- Margie Thickstun, Carolyn Mascaro (Lit. & Creative Writing) - Assisted with box permanent location.
- Mark Kinne, Bill Huggins (Physical Plant) - Provided keys for permanent location.

# Sandbox Calibration (and some installation)

## Hardware Calibration

Calibrate and configure the devices (after installation of software on the computer).

### 1.) Get Intrinsic Calibration Parameters

Plug in your first-generation Kinect device and download intrinsic calibration parameters directly from its firmware. In a terminal window, run:

```
sudo /usr/local/bin/KinectUtil getCalib 0
```

Note: This might ask you for your password again; if so, enter it (arsandbox) to continue.

### 2.) Physically Align Camera

Align your camera so that its field of view covers the interior of your sandbox. Use RawKinectViewer to guide you during alignment. To start it, run in a terminal window:

```
sudo /usr/local/bin/RawKinectViewer -compress 0
```

While looking at the camera image in RawKinectViewer make sure the Kinect sensor is lined up with the sides of the sandbox. If the sandbox appears off from the viewport of the Kinect camera, tilt the Kinect camera until you can get these aligned.

### 3.) Calculate Base Plane

This step will calculate the plane (base level) of the sandbox, which is the height of the sand when it's perfectly flat.

If your box has sand in it put a sheet of cardboard or other flat surface over the top of the box. If you use this method, remember to measure the distance (in cm) from the cardboard to the bottom of the sandbox. In this case, the bottom of the box was ~16 cm below the cardboard. When you get the base plane equation from the rest of this step remember to add the value, from your measurement, to the last number in the equation. It's all explained in the following video:

AR Sandbox Calibration - this video shows steps 3 through 5 (https://www.youtube.com/watch?v=EW2PtRsQQr0).

Calculate your sandbox’s base plane, by following the instructions in the AR Sandbox Calibration video (above) that shows all required calibration steps in one. You can use the already-running instance of RawKinectViewer.

## 4.) Enter Equation into BoxLayout.txt

You need to enter the base plane equation (and the 3D sand surface extents in the next step) into the BoxLayout.txt file in /usr/local/etc/SARndbox-2.3

```
cd /usr/local/etc/SARndbox-2.3
```

```
kwrite BoxLayout.txt
```

Now enter the base plane equation as described in the video. To copy text from a terminal window, highlight the desired text with the mouse, and then either right-click into the terminal window and select “Copy” from the pop-up menu that appears, or press Shift-Ctrl-c. To paste into the text editor, use the “Edit” menu, or press Ctrl-v.

The equation will look something like the following from the RawConnectViewer output in the terminal:

```
 (-0.0076185, 0.0271708, 0.999602) = -98.0000
```

Be sure to replace the = with a comma and add the measured value from the cardboard to the bottom of the sandbox from above. The last number in this sequence is the depth. So in our example -98 becomes -114. So the first line of our BoxLayout.txt file has this:

```
(-0.0076185, 0.0271708, 0.999602), -114.0000
```

## 5.) Measure 3D extents of the Sand Surface

Note: Be sure to remove the cardboard covering the sandbox from the previous step.

In the newly-released Kinect-3.2 package, this can be done inside RawKinectViewer as well by following the instructions in this video (https://youtu.be/EW2PtRsQQr0?t=4m10s) starting at 4:10. Make sure to measure the box corners in the order lower-left, lower-right, upper-left, upper-right. After you have copied the box corner positions into the text editor as described in the video, save the file (via the “File” menu or by pressing Ctrl-s), and quit from the text editor (via the “File” menu or by pressing Ctrl-q).

In the terminal window from where you executed the RawConnectViewer you'll have the following four corner points:

```
(  -48.6846899089,  -36.4482382583,    -94.8705084084)
(   48.3846899089,  -34.3992382583,    -89.3885084084)
(  -50.6746899089,   35.8072382583,    -97.4085084084)
(   48.7946899089,   36.4782382583,    -91.7415084084)
```

Add these to the BoxLayout.txt file after the plane equation from the last step. No changes need to be made to the formatting from the terminal.

You can now exit RawKinectViewer and save BoxLayout.txt

## 6.) Align Projector

Align your projector such that its image fills the interior of your sandbox. You can use the calibration grid drawn by Vrui's XBackground utility as a guide. In a terminal, run:

```
/usr/local/bin/XBackground
```

After the window showing the calibration grid appears, press the "F11" key to toggle it into full-screen mode. Ensure that the window really covers the entire screen, i.e., that there are no title bar, desktop panel, or other decorations left.

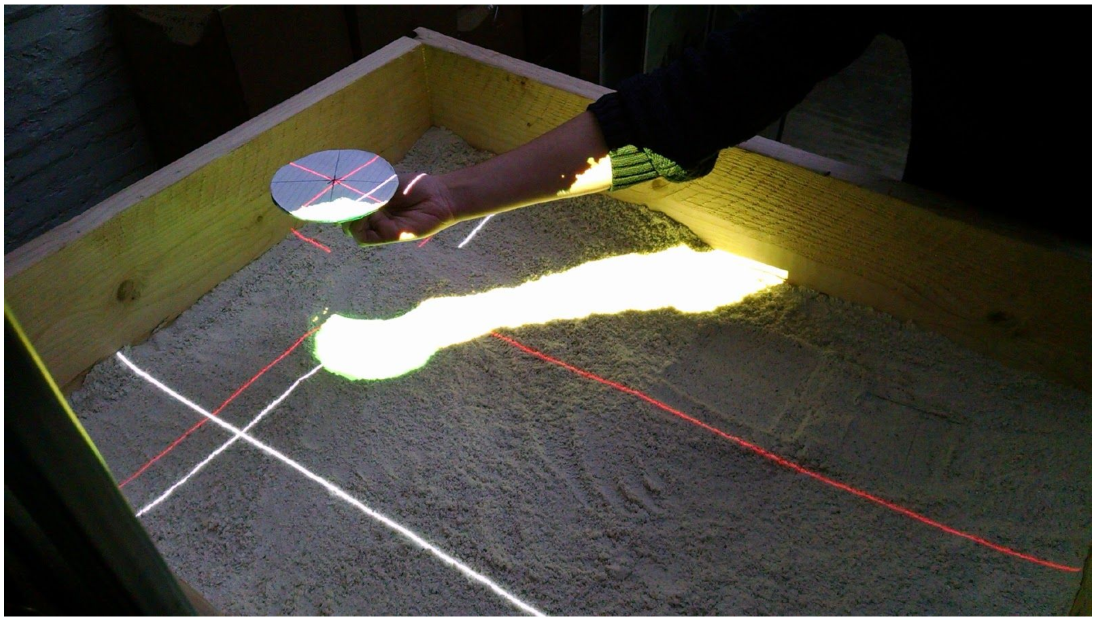
Press Esc to close XBackground's window when you're done.

## 7.) Calibrate the Projector

Determine the resolution that your projector is running at. In our case it is running at 1024 x 800. You will also need to have created the calibration tool. This consists of:

- CDROM Disk
- Coat Hanger
- blank sheet of paper

Create the calibration tool. This is made up of a CD disc with blank sheet of paper glued to it. On the paper you have two lines perpendicular to one another, one horizontally and one vertically so that they make up a + sign with a point in the center of the disc. Using a wire coat hanger proved to work the best to form a handle to use when taking reference points.



Calibrate the Kinect camera and the projector with respect to each other by running the CalibrateProjector utility:

```
/usr/local/bin/CalibrateProjector -s <width> <height>
```

where are the width and height of your projector's image in pixels. For example, for an XGA projector like the recommended BenQ, the command would be:

```
/usr/local/bin/CalibrateProjector -s 1024 800
```

Very important: switch CalibrateProjector's window to full-screen mode by pressing F11 before proceeding. Then follow the instructions in this video (https://youtu.be/EW2PtRsQQr0?t=10m10s) starting at 10:10.

Create a capture tool and bind it to the "2" key. This will allow you to re-capture a background image after you've changed the topography of the sand.

Pressing the "1" key will take each reference point by lining up your crosshairs from the CD to the lines drawn in the sandbox.

## 8.) Run AR Sandbox

Finally, run the main AR Sandbox application:

```
/usr/local/bin/SARndbox -uhm -fpv &
```

# SARndbox Overview

**Summary of SARndbox process**

"Raw depth frames arrive from the Kinect camera at 30 frames per second and are fed into a statistical evaluation filter with a fixed configurable per-pixel buffer size (currently defaulting to 30 frames, corresponding to 1 second delay), which serves the triple purpose of filtering out moving objects such as hands or tools, reducing the noise inherent in the Kinect's depth data stream, and filling in missing data in the depth stream.

The resulting topographic surface is then rendered from the point of view of the data projector suspended above the sandbox, with the effect that the projected topography exactly matches the real sand topography.

The software uses a combination of several GLSL shaders to color the surface by elevation using customizable color maps (the default color map used right now was provided by M. Burak Yikilmaz, a post-doc in the UC Davis Department of Geology), and to add real-time topographic contour lines."
- UC Davis SARndbox Documentation

**Some vocab**

- vector space
  - aka linear space; a collection of objects called vectors, which may be added together and multiplied ("scaled") by numbers called scalars
- linear transformation
  - aka linear map; a mapping between two modules (including vector spaces) that preserves the operations of addition and scalar multiplication
- transformation matrix
  - matrix representation of a linear transformation
- projective transformation matrix
  - non-linear transformations on an n-dimensional Euclidean space R^n that can be represented as linear transformations on the n+1-dimensional space R^n+1
  - 4×4 transformation matrices are widely used in 3D computer graphics
- mipmap
  - mipmaps or pyramids are pre-calculated, optimized sequences of images, each of which is a progressively lower resolution representation of the same image
  - the height and width of each image, or level, in the mipmap is a power of two smaller than the previous level
  - intended to increase rendering speed and reduce aliasing artifacts
- texture map

    - image applied (mapped) to the surface of a shape or polygon; may be a bitmap image or a procedural texture
    - may have 1-3 dimensions, although 2 dimensions are most common for visible surfaces
    - rendering APIs typically manage texture map resources as buffers or surfaces, and may allow 'render to texture' for additional effects such as post processing, environment mapping
    - usually contain RGB color data (either stored as direct color, compressed formats, or indexed color), and sometimes an additional channel for alpha blending (RGBA)
- pixel shader
    - also known as fragment shaders, these compute color and other attributes of each "fragment" - a technical term usually meaning a single pixel
    - the simplest kinds output one screen pixel as a color value; more complex shaders with multiple inputs/outputs are also possible
- geometry shader
    - can generate new graphics primitives, such as points, lines, and triangles, from those primitives that were sent to the beginning of the graphics pipeline
- GLSL
    - OpenGL Shading Language for writing shaders
- OpenGL
    - Open Graphics Library (OpenGL) is a cross-language, cross-platform application programming interface for rendering 2D and 3D vector graphics
    - typically used to interact with a graphics processing unit to achieve hardware-accelerated rendering

## Kinect Overview

This library is part of the original SARndbox project. It manages the XBox Kinect camera that is mounted over the sandbox and reads in the depth information.

Kinect files directly included in the SARndbox code:

FrameBuffer - Class for reference-counted decoded color or depth frame buffers.

Camera - Wrapper class to represent the color and depth camera interface aspects of the Kinect sensor.

FrameSource - Base class for objects that create streams of depth and color frames.

FileFrameSource - Class to stream depth and color frames from a pair of time-stamped depth and color stream files.

DirectFrameSource - Intermediate class for frame sources that are directly connected to a camera device.

OpenDirectFrameSource - Helper functions to open a 3D camera by index or serial number without having to know its type.

# Vrui Overview

This library is part of the original SARndbox project. Its purpose in SARndbox is to manage the graphical aspects of the program, including file handling, color map creation, and GUI creation.

**Summary of purpose**
"The Vrui VR toolkit aims to support fully scalable and portable applications that run on a range of VR environments starting from a laptop with a touchpad, over desktop environments with special input devices such as space balls, to full-blown immersive VR environments ranging from a single-screen workbench to a multi-screen tiled display wall or CAVE. **Applications using the Vrui VR toolkit are written without a particular input environment in mind, and Vrui-enabled VR environments are configured to map the available input devices to application functions** such that the application appears to be written natively for the environment it runs on." - UC Davis Vrui Documentation

**Tiny Example**
A complete Vrui "Hello World" application, rendering a cube and a sphere with full viewpoint navigation, can be written in 34 lines of code.

**Some Notices**
Do not perform OpenAL or OpenGL context setup

A Vrui application is not the only party rendering to its graphics or sound contexts in the Vrui runtime environment. Other parties include the tool graph, the GUI widget manager, any loaded vislets, etc. **It is the responsibility of the Vrui toolkit, not the developer, to set up and initialize OpenGL or OpenAL contexts at the beginning of the rendering cycle.** Application rendering code is called when all rendering contexts are already set up properly, and other 3D geometry or sound might already have been rendered into them. It is the developer's responsibility not to change context state permanently, and not to destroy what has already been rendered, or will be rendered.

Concretely, **applications must never change the projection matrix, clear the color or depth buffers, change the viewport or depth range, leave texture objects or buffers bound, etc.** If some functionality requires to do any of these, it is the developer's responsibility to ensure that the larger Vrui runtime environment is not affected, and that all state changes are properly restored before an application's rendering functions return.

Do understand the relationship between physical and navigational space

Vrui works in two coordinate spaces: physical space and navigational space. Navigational space plays the same role as model space in other 3D toolkits; it is the space in which application-specific data lives. Navigation space can have arbitrary, application-defined orientations and measurement units. Physical space, on the other hand, is bound to the display hardware of a particular VR environment; it is the space in which a user lives. It can still have arbitrary orientations and measurement units, but it expresses the layout of a VR environment in the real world.

The mapping between physical space and navigational space is expressed by the navigation transformation, which has a similar purpose as the modelview matrix in typical OpenGL applications. The navigation transformation is also the means how Vrui changes viewpoints, or how users navigate through their 3D data (hence the name). Any changes to the viewpoint in Vrui applications are expressed by changes to the navigation transformation. This is typically done by dedicated navigation tools, which are generalizations of the viewpoint manipulation metaphors typically used in desktop 3D applications, such as virtual trackballs.

## Software Information

- Original Vrui is located at https://github.com/KeckCAVES/Vrui
- Original Kinect is located at https://github.com/KeckCAVES/Kinect
- Original SARndbox is located at https://github.com/KeckCAVES/SARndbox
- Complete installation instructions for SARndbox
- Books of Sand code is located at https://github.com/booksofsand

### Installation Notes

- The arsandbox user password is "arsandbox"
- The code libraries are installed in /opt and are git repositories connected to the booksofsand Github organization
- The applications are compiled in /opt and then installed into the /usr/local directories
- The user arsandbox owns the files in /opt and /usr/local, so that account has full control over these installations to make changes as needed
- SELinux and Firewall are disabled so remember this isn't as secure as it could be if you leave it plugged into the network
- The user arsandbox is also a sudoer, meaning if you need root access you can become root using the following command: `sudo su -`

### Usage Notes

- Vrui and Kinect both have two git branches: `master` and `flow`
    - The `master` branches contain the original, unchanged code
    - The `flow` branches contain print statements (mostly of the form of "In FrameBuffer (Kinect::FrameBuffer.cpp).") added for the purpose of tracing the actions of the SARndbox program
    - To switch from one branch to the other, follow these steps for each library:
        - `cd /opt/[libraryname]`
        - `git checkout [branchname]`
        - `make`
        - `make install`
        - 
        - `cd /opt/Vrui             // for example`
        - `git checkout master`
        - `make`
        - `make install`
- BooksOfSand has four git branches: `master`, `original`, `app`, and `images`
    - The `master` branch is similar to the Vrui and Kinect flow branches in purpose and implementation
    - The `original` branch is the original SARndbox code

    - The `app` branch contains work on combining the Graphics code with the SARndbox code with SARndbox as the base instead of Qt
    - The `images` branch contains work on using images of book pages instead of plain text files within the SARndbox code
    - To switch from one branch to another, follow these steps:
        - `cd /opt/BooksofSand`
        - `git checkout [branchname]`
        - `make`
        - `make install`
        - 
        - `cd /opt/BooksofSand       // for example`
        - `git checkout original`
        - `make`
        - `make install`
- Graphics has two git branches: `master` and `demo`
    - The `master` branch contains all work in progress, including attempting to connect with Kinect in the kinecthandler code
    - The `demo` branch contains simplified code just to demonstrate the graphical component; has no connection to real-time depth maps
    - To switch from one branch to another, follow these steps:
        - `cd /opt/Graphics`
        - `git checkout [branchname]`
        - `make`
- To run the original SARndbox program
    - Make sure the `master` branches of Vrui and Kinect are installed
    - Make sure the `original` branch of BooksOfSand is installed
    - In a terminal, run: `sudo /usr/local/bin/SARndbox -fpv -uhm`
    - Move the SARndbox window to the sandbox (as if it were a second monitor) and full-screen it with F11
- To run the BooksOfSand program
    - Compile Graphics by running `make` in the /opt/Graphics directory
    - For prose-style (where a single source text is used to produce the entire 3D display of text), run: `./Graphics [file]`
        - e.g.: `./Graphics texts/rowling.txt`
    - For poetry-style (where multiple source texts are used, with one file per layer of displayed text), run: `./Graphics [file1] [file2] ...`
        - e.g.: `./Graphics texts/poe1.txt texts/poe2.txt`
- Copies of the original BoxLayout.txt and ProjectorMatrix.dat are saved on the computer in /opt/BooksOfSand/etc/SARndbox-2.3, named with ‘orig’ prepended
    - Made new BoxLayout.txt and ProjectorMatrix.dat files during our calibration
    - If SARndbox topography projection seems to be flipped (i.e. moving sand on one side changes the other side), then likely the ProjectorMatrix.dat in /usr/local/etc/SARndbox-2.3 has been replaced with the original

    - If most of the sand is blue, the base depth in BoxLayout.txt is too high

## Connecting Qt, Kinect, and Vrui

Once we decided to try making the graphical application in Qt, one challenge was figuring out how to connect the GUI with the Kinect camera. The Kinect and Vrui libraries are interconnected, so we needed to be able to compile Qt code with those libraries. We tried a couple approaches: using the SARndbox makefile as a foundation, and using the Qt auto-generated makefile as a foundation.

We were able to get the code to compile properly using the second approach; we just had to add some Vrui references. View Makefile in Graphics for details. Basics below:

```
INCPATH     += -I/usr/local/include/Vrui-4.2
LFLAGS      += -L/usr/local/lib64/Vrui-4.2 -Wl
LIBS        += -lKinect.g++-3 -lMisc.g++-3 -lIO.g++-3
#    e.g. -lKinect.g++-3 represents
#         /usr/local/lib64/Vrui-4.2/libKinect.g++-3.so
```

Here's some potentially useful sources for future work with compilation, if needed:

With Qt makefile
Qt 5 Creating Project Files
Qt 5 Creating Project Files - Declaring Other Libraries
Qt 5 Third Party Libraries
SO qmake Project Dependencies

With SARndbox makefile
Linux Qt Makefile
Makefile to build QT projects (Linux) without qmake

## Graphics (text manipulation)

The Graphics repository is the graphical component to the Books of Sand project. Uses Qt (C++ graphical application framework) to display text on a screen; updates displayed text based on simplified depth maps. Made to replace SARndbox repository. Still needs work to connect to Kinect to receive depth maps in real time.

**Main**

- Collects arguments passed on the command line and assumes they're file names (files to be opened and used as text sources)
- Initializes a sandboxwindow object
- Begins Qt application execution loop to keep application running

**Sandboxwindow**

- <u>Data structure</u>
- Stores source text in sourceText 3D array of strings, where each string is one letter at a unique row, column, and depth
- Tracks which letters to display with depthsDisplayed 2D array of depth level numbers
  - e.g. if depthsDisplayed[0][5] equals 2, then row 0, col 5 of text being graphically displayed is the letter at sourceText[2][0][5]
- Displays text by building one string per row of text, selecting each letter from the sourceText 3D array based on the depth to be displayed at the current row and column
- Currently has maximums set for number of rows, columns, and depths
- 
- <u>Input</u>
- Reads in plain text from files to build array of source text
  - If passed one file name, reads in text and fills the 3D array beginning at top level depth and "wrapping around" to the next depth as the text fills each layer
    - e.g. reads in text from fileA and fills up every row and column at depth 0, continues reading from fileA to fill up depth 1, etc. until the max depth is hit or reaches EOF
  - If passed multiple file names, assigns each file to a depth
    - e.g. fileA fills depth 0 in the 3D array; fileB fills depth 1…
- 
- <u>Updating</u>
- Updates graphical display letter by letter
  - Accepts a depthsToDisplay 2D array containing modified depth levels in same format as depthsDisplayed array
  - Updates depthsDisplayed array and graphical display to correspond to new depths

    - Requires that depth map is simplified elsewhere to produce depthsToDisplay array containing depth level integers as opposed to actual depth float values

**Kinecthandler**

- <u>Purpose</u>
- Supposed to bridge the gap between the depth map read in from the XBox Kinect and the sandboxwindow application
- Contains code from SARndbox (Sandbox.cpp and Sandbox.h) in an attempt to connect to Kinect and read in a proper depth map
- 
- <u>Needs work</u>
- The depth map we received sometimes only had half a map (stored as float values in array), and the values were significantly different from those we would receive when running the original SARndbox application
- Once the depth map can be read in correctly, need to average groups of depth points to determine approximate depth of sand at the location of each displayed letter (currently the Kinect produces a depth map with 640 x 480 points)
- Then convert the depth averages to depth levels to pass to the sandboxwindow for a graphical update

# **OpenGL Context Data**

Brief description of OpenGL Context Data (used in SARndbox).

[UC Davis Context Data Explanation](UC Davis Context Data Explanation)
See IndexedTriangleSet example.

"The approach embodied by GLContextData/GLObject is to separate per-application state from per-context state, and to provide a mechanism to associate per-context state with an application when that state is needed for rendering."

"Since it is not allowed to change application state from inside a render() method, an application can only create new objects from some other method, for example an event callback. That means that per-context state must be initialized right before an object is rendered first in each context it is rendered in."

- Any class that has per-context state must be derived from GLObject.
- Any per-context state of the class must be separated into an embedded DataItem structure derived from GLObject::DataItem.
- The DataItem constructor allocates OpenGL resources (texture objects, vertex buffers, etc.), but does not necessarily have to initialize those resources.
```
IndexedTriangleSet::DataItem::DataItem(void)
     :textureObjectId(0)
     {
     glGenTextures(1,&textureObjectId);
     }
```
- The virtual DataItem destructor releases all allocated OpenGL resources.
```
IndexedTriangleSet::DataItem::~DataItem(void)
     {
     glDeleteTextures(1,&textureObjectId);
     }
```

The IndexedTriangleSet initContext() method creates a data item, stores it in the GLContextData, and initializes the OpenGL resources.
```
void IndexedTriangleSet::initContext(GLContextData& contextData) const
      {
      /* Create a new data item: */
      DataItem* dataItem=new DataItem();

      /* Associate object and data item in GLContextData: */
      contextData.addDataItem(this,dataItem);

      /* Read and upload texture image into dataItem->textureObjectId: */
      glBindTexture(GL_TEXTURE_2D,dataItem->textureObjectId);
      ...
      /* Protect texture object: */
      glBindTexture(GL_TEXTURE_2D,0);
      }
```

The IndexedTriangleSet render() method retrieves the data item from the GLContextData, and uses it to render.

```
void IndexedTriangleSet::render(GLContextData& contextData) const
      {
      /* Retrieve data item from GLContextData: */
      DataItem* dataItem=contextData.retrieveDataItem<DataItem>(this);

      /* Activate texture object: */
      glBindTexture(GL_TEXTURE_2D,dataItem->textureObjectId);

      /* Render triangles: */
      ...

      /* Protect texture object: */
      glBindTexture(GL_TEXTURE_2D,0);
      }
```

# Texture Mapping

Brief description of texture mapping and how it works with OpenGL (used in SARndbox).

Texture mapping is applying any type of picture on one or more faces of a 3D model.
http://ogldev.atspace.co.uk/www/tutorial16/tutorial16.html

Just like other objects, textures have to be bound to apply operations to them. **Since images are 2D arrays of pixels, it will be bound to the GL_TEXTURE_2D target.**
https://open.gl/textures

1. Load a texture into OpenGL.
2. Supply texture coordinates with the vertices of the object to which the texture should be mapped.
3. Perform a sampling operation from the texture using the texture coordinates in order to get the pixel color (the result of sampling is a texel. a pixel in a texture).

The method by which the final texel value is selected is known as 'filtering'. The simple approach of rounding the texture location is known as 'nearest filtering' and the more complex approach that we saw is called 'linear filtering'. Another name for nearest filtering you may come across is 'point filtering'. OpenGL supports several types of filters and you have the option to choose.

Texturing in OpenGL means manipulating the intricate connections between four concepts: the texture object, the texture unit, the sampler object and the sampler uniform in the shader.

- The texture object contains the data of the texture image itself, i.e., the texels.
- The texture object is not bound directly into the shader (where the actual sampling takes place). Instead, it is bound to a 'texture unit' whose index is passed to the shader. So the shader reaches the texture object by going through the texture unit. When you bind a texture object to a texture unit you specify the target (1D, 2D, etc).
- The sampling operation (usually) takes place inside the fragment shader.
- You can also create a sampler object, configure it with a sampling state and bind it to the texture unit.

OpenGL knows how to load texture data in different formats from a memory location but does not provide any means for loading the texture into memory from image files such as PNG and JPG. We are going to use an external library for that. (**Vrui provides this loading with its image classes!**)

# OpenGL Functions for Textures

A nonexhaustive listing of OpenGL functions used in the SARndbox code.

## OpenGL Refpages

### glBindTexture

This function tells OpenGL the texture object we refer to in all the following texture related calls, until a new texture object is bound. Specify the texture target, which can be GL_TEXTURE_1D, GL_TEXTURE_2D, etc. (target of the active texture unit to which the texture is bound) and the handle (the name of a texture).

```
void glBindTexture(
      GLenum target,
      GLuint texture);
```

e.g. in BooksOfSand/SurfaceRenderer.cpp:

```
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,dataItem->contourLineColorTextureO
bject);
```

### glTexParameteri

glTexParameteri args: texture target, parameter name, parameter value.

```
void glTexParameteri(
      GLenum target,
      GLenum pname,
      GLint param);
```

e.g. in BooksOfSand/SurfaceRenderer.cpp:

```
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,GL_TEXTURE_MAG_FILTER,GL_NEAREST
);
```

### glTexImage2D

glTexImage2D args: texture target, level of detail (mipmap; 0 is highest resolution), internal format (e.g. GL_RGBA), width (texels), height (texels), border (0 is none), source format, source type, source memory address.

```
void glTexImage2D(
      GLenum target,
      GLint level,
      GLint internalFormat,
      GLsizei width,
      GLsizei height,
      GLint border,
      GLenum format,
      GLenum type,
        const GLvoid * data);
```

e.g. in BooksOfSand/SurfaceRenderer.cpp:

```
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,0,GL_R32F,dataItem->contourLineFram
ebufferSize[0],dataItem->contourLineFramebufferSize[1],0,GL_LUMINANCE,GL_
UNSIGNED_BYTE,0);
```

## glActiveTexture

Specifies which texture to make active (active texture is used when displaying) using a texture ID.

```
void glActiveTexture(
        GLenum texture);
```

e.g. in BooksOfSand/SurfaceRenderer.cpp:

```
/* Bind the height color map texture: */
glActiveTextureARB(GL_TEXTURE1_ARB);
```

## glDrawElements

Renders multiple primitives from an array. Args: what kind of primitives to render, number of elements to render, type of the values in indices (GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT), and pointer to the location where the indices are stored.

```
void glDrawElements(
        GLenum mode,
        GLsizei count,
        GLenum type,
        const GLvoid * indices);
```

e.g. in BooksOfSand/DepthImageRenderer.cpp:

```
/* Draw the surface template: */
GLVertexArrayParts::enable(Vertex::getPartsMask());
glVertexPointer(static_cast<const Vertex*>(0));
GLuint* indexPtr=0;
for(unsigned int y=1; y<depthImageSize[1];
    ++y, indexPtr+=depthImageSize[0]*2)
      glDrawElements(GL_QUAD_STRIP,depthImageSize[0]*2,
                     GL_UNSIGNED_INT,indexPtr);
GLVertexArrayParts::disable(Vertex::getPartsMask());
```

## glBindBufferARB

Old name for glBindBuffer. Binds a buffer object to the specified buffer binding point. Entering 0 for the buffer object is for unbinding, like binding to a null buffer.

```
void glBindBuffer(
        GLenum target,
        GLuint buffer);
```

e.g. in BooksOfSand/DepthImageRenderer.cpp:

```
glBindBufferARB(GL_ARRAY_BUFFER_ARB,dataItem->vertexBuffer);
```

## glBegin, glEnd

glBegin specifies how to interpret the vertices that follow the glBegin statement and precede the glEnd statement.

```
void glBegin(
        GLenum mode);

void glEnd(void);
```

e.g. in Vrui/ExamplePrograms/ImageViewer.cpp:

```
glBegin(GL_QUADS);
// ^ specifies the following vertices as groups of 4 to interpret as
quadrilaterals
...
glEnd();
// ^ ends the listing of vertices
```

## glTexCoord

Sets the current texture coordinates (sometimes called after glBegin() when specifying vertices). Multiple versions for different argument types. Texture coordinates specify the point in the texture image that will correspond to the vertex you are specifying them for. (See helpful SO answer.)

e.g. for 2D coordinates in floats:

```
void glTexCoord2f(
        GLfloat s,
        GLfloat t);
```

e.g. in Vrui/ExamplePrograms/ImageViewer.cpp:

```
const GLfloat* texMin=tex.getTexCoordMin();
const GLfloat* texMax=tex.getTexCoordMax();
glBegin(GL_QUADS);
glTexCoord2f(texMin[0],texMin[1]);
```

## glVertex

Specifies a vertex after glBegin(). Multiple versions for different argument types.

e.g. for 2D vertex in ints:

```
void glVertex2i(
        GLint x,
        GLint y);
```

e.g. in Vrui/ExamplePrograms/ImageViewer.cpp:

```
glBegin(GL_QUADS);
glTexCoord2f(texMin[0],texMin[1]);
glVertex2i(0,0);
```

## glUniform

Specifies the value of a uniform variable for the current program object. Multiple versions for different argument types.

glUniform modifies the value of a uniform variable or a uniform variable array. The location of the uniform variable to be modified is specified by location, which should be a value returned by glGetUniformLocation. glUniform operates on the program object that was made part of current state by calling glUseProgram.

glUseProgram is called in DepthImageRenderer.cpp and SurfaceRenderer.cpp

The commands glUniform{1|2|3|4}{f|i|ui}v can be used to modify a single uniform variable or a uniform variable array. The number specified in the command should match the number of components in the data type of the specified uniform variable (e.g., 1 for float, int, unsigned int, bool; 2 for vec2, ivec2, uvec2, bvec2, etc.). The suffix f indicates that floating-point values are being passed.

```
void glUniform4fv(
      GLint location,
      GLsizei count,
      const GLfloat *value);
```

e.g. in ElevationColorMap:

```
GLfloat texturePlaneEq[4]; // in .h
...
void ElevationColorMap::uploadTexturePlane(GLint location) const {
    glUniformARB<4>(location, 1, texturePlaneEq);
}
```

## Uniform

A uniform is a global GLSL variable declared with the "uniform" storage qualifier. These act as parameters that the user of a shader program can pass to that program. They are stored in a program object. Uniforms are so named because they do not change from one execution of a shader program to the next within a particular rendering call. This makes them unlike shader stage inputs and outputs, which are often different for each invocation of a program stage.

## glMatrixMode

glMatrixMode sets the current matrix mode, which can assume one of four values:
GL_MODELVIEW: Applies subsequent matrix operations to the modelview matrix stack.
GL_PROJECTION: Applies subsequent matrix operations to the projection matrix stack.
GL_TEXTURE: Applies subsequent matrix operations to the texture matrix stack.
GL_COLOR: Applies subsequent matrix operations to the color matrix stack.

e.g. in CalibrateProjector.cpp:

```
glMatrixMode(GL_PROJECTION); // called in display before a bunch of code
glMatrixMode(GL_MODELVIEW); // called at end of display for no purpose I can see
```

SO answer on the purpose of modelview projection