Technical Memorandum

Applied Space Exploration Laboratory (ASEL)
Benjamin M. Statler College of Engineering and Mineral Resources
West Virginia University

Date: 9 October 2014
Memo: ASEL–14–005

To: T. Evans (WVU), M. Strube (NASA/GSFC), J. VanEepoel (NASA/GSFC), C. D'Souza (NASA/JSC)
From: Dr. John O. Woods & Dr. John A. Christian (WVU)

Title: A real-time, software-based 3D sensor simulator

1 Introduction

This memorandum introduces Glidar, an OpenGL-based simulator of 3D sensor systems, written in C++. Glidar works in real-time, utilizing modern graphics processing unit (GPU) architecture to produce point clouds. OpenGL, which stands for Open Graphics Library, is an application programming interface (API) widely used for rendering three-dimensional scenes into two-dimensional images; it is common in video games, as a variety of operating systems include OpenGL libraries.

Glidar was originally intended for the quick production of sensor data needed to test point cloud processing algorithms. While test clouds could have been produced with the lab's SwissRanger 4000 time-of-flight camera, Glidar has the advantage of being able to render any object for which a 3D model is available — without the need for expensive mockups, test setups, and robotic solutions.

Glidar does not and cannot simulate any specific piece of hardware. It can, however, be used in lieu of nearly any sensor which produces a point cloud representation of an object — including Microsoft Kinect, time-of-flight cameras, lidars, and stereo cameras. It permits point clouds to be saved as files, visualized, or published in a publish–subscribe architecture.

While other groups have utilized graphics hardware to produce radar and lidar sensor simulators in the past,[?, ?, ?] to our knowledge, none of these have been released for use by other groups. It is our hope that Glidar may be released on GitHub under an open source software license. Allowing others to view and modify the code would dramatically improve the set of features available in the software, and would help make others aware of the work being done at the West Virginia University.

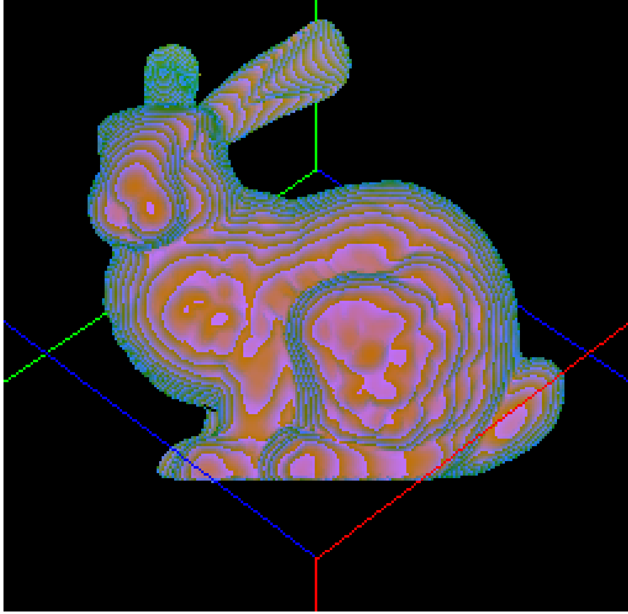
2 Hardware and Software Requirements

Glidar makes use of a number of open source libraries — including the Open Asset Import Library (Assimp), for loading 3D models; Magick++ and ImageMagick for model textures; CMake, for compiling; and glew and glfw for cross-platform interoperability of OpenGL components. It also uses the Point Cloud Library, PCL, for parsing command line options — used for convenience and for interoperability for other parts of our testing framework, but easily replaceable. For publishing point clouds (*i.e.*, publish–subscribe), ZeroMQ is needed.

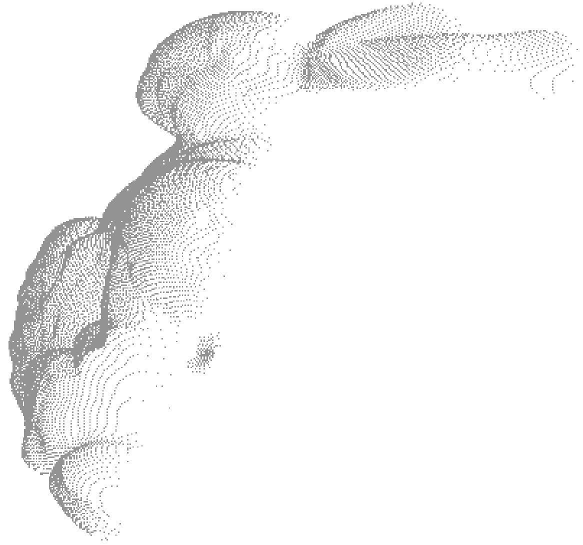
Glidar requires graphics hardware supporting at least glsl 1.20 — a standard introduced in 2006, and likely available with any modern graphics card.

3 Inner Workings

Most of Glidar consists of a fairly standard pattern for an OpenGL application. The key innovations are in the glsl shaders; the vertex and fragment shaders are derived from those used for a fairly standard spotlight, and most Glidar-specific maths are found in the fragment shader.



(a) GLIDAR screenshot.



(b) Point cloud output.

Figure 1: GLIDAR is capable of providing sensor images in the form of point clouds, which can be used to test machine learning strategies involving point clouds. Shown is a screenshot of GLIDAR's rendering of the Stanford bunny (a) and the resulting point cloud visualized using `pcl_viewer` (b).

Specifically, for each pixel, the light intensity is returned in the red channel, and the green and blue channels carry depth information for that pixel (Fig. 1). The alpha channel is currently unused. Light intensity is calculated using the standard OpenGL lighting model,

$$a = \left(a_{\text{const}} + a_{\text{lin}} d + a_{\text{quad}} d^2\right)^{-1} , \tag{1}$$

where $d = \vec{l}$ is the object–sensor distance and a is the attenuation. a_{const}, a_{lin}, and a_{quad} are the constant, linear, and quadratic attenuation values for the light source, respectively. Red intensity

$$r = ac \tag{2}$$

is then calculated from the color,

$$c = \left(c_{\text{a}} + c_{\text{d}} \max(\vec{n} \cdot \|\vec{l}\|, 0) + c_{\text{s}} \max(\vec{n} \cdot \vec{h}, 0)^s\right) \tag{3}$$

with $\vec{n}$ as the surface normal at the pixel being sampled, $\vec{h}$ as the normalized half-vector of the light, and c_{d}, c_{a}, and c_{s} representing the diffuse, ambient, and specular color information (which depend on both the light source and the object's textures and materials), respectively. The material's *shininess* (like reflectance, but graphics pipeline-specific) is described by s. Values for light attenuation were chosen arbitrarily, and do not describe any specific sensor.

A noise generator is also included in the fragment shader, but our group has not had cause to utilize it yet. While textures are currently utilized for light intensity calculations, a key future goal is to offer an option which permits varying the noise according to bump and shininess textures from the model in order to simulate multi-layer insulation and other complex materials.

4 Future Improvements

At present, GLIDAR has limited ability to move with respect to a rendered model. The camera can be moved only along its boresight (the z axis), and the sensed object may be set spinning; but more complicated trajectories which involve movement along the x and y axes are not yet implemented.

5 Usage Notes

The codes for Glidar is stored in a `git` repository on the provided media. A local copy of the repository is needed, as the codes must be compiled, and it may be copied either with `cp` or `git clone`.

To compile the application, a build directory must be created:

```
cd glidar/
mkdir build
cd build/
cmake ..
```

If `cmake` succeeded, the application may now be built using `make`.

Glidar should be run in the root of its source tree (not from the build directory), via command line, `e.g.`,

```
build/glidar ISS\ models\ 2011/Objects/Modules/MLM/MLM.lwo \
  --scale 0.24 --dr 0.1,0.01,0 --r 0,0,0 \
  --camera-z 1000 -w 256 -h 256 --fov 20 \
  -p 65431 --pub-rate 15 --subscribers 1
```

Note that on a Mac system, the binary will most likely be `build/glidar.app/Contents/MacOS/glidar`, not `build/glidar`.

Command line parameters are as follows:

- The first argument must be the model filename.
- `--scale`: The amount by which to scale the model.
- `--dr`: Rate of rotation about x, y, and z axes, respectively.
- `--r`: Initial attitude of the 3D model.
- `--camera-z`: Initial distance between the camera and the object (typically in meters, though the example above is decimeters.
- `-w`, `-h`: Width and height of the sensor.
- `--fov`: Sensor field-of-view in degrees.
- `-p` or `--port`: Output port to publish point clouds to.
- `--pub-rate`: Rate at which to publish point clouds and poses (every n loop iterations).
- `--subscribers`: Number of subscribers to wait for before beginning to publish.
- `--pcd`: Filename to which to save the first rendered point cloud (exiting immediately). Do not use in combination with publishing.

Additionally, the camera position can be controlled manually using the plus and minus keys on the keyboard. The `s` key will save a point cloud to the file `buffer.pcd`. Saving, either through the `--pcd` option or the `s` key, also produces a `.transform` file which describes the model rotations and the camera translation.

Glidar may be terminated via the `esc` key.

5.1 Publishing Format

Two types of data are published by Glidar: point clouds and poses (in the form of 4×4 homogeneous transform matrices).

The type of data being transmitted is indicated by the first byte's ASCII value. '`c`' indicates a point cloud, and '`p`' indicates a pose. For clouds, the next piece of information will be the number of points n (type `size_t`), followed by $4n$ floating point values. Every four values describe a single point (x, y, z, and intensity). For pose messages, sixteen floats will be transmitted.

When Glidar is exited normally, it publishes two additional messages of eight bytes — one to those subscribers looking for cloud messages and the other to those awaiting poses. The last seven bytes of each of these messages will be `KTHXBAI`, and indicates that subscribers should no longer wait for new messages.

6 Conclusions

In machine learning tasks, it is often preferable to use automatically generated data — or recorded information — to fine-tune a classifier, recognizer, or filter. Generating these data requires no expensive hardware and no difficult-to-obtain mockups. Furthermore, in a traditional setup, a robot arm would likely be required to manipulate the sensor in order that accurate camera position and attitude information could be made available. GLIDAR, however, removes the need for all of these complications.

Acknowledgments

The authors would like to thank Jordan Sell and Andrew Rhodes of West Virginia University (WVU) for reviewing this manuscript and providing critical support along the way. This research was made possible by contract XXXXXX.