Journal of
Imaging

MDPI

Short Note

GLIDAR: An OpenGL-based, Real-Time, and Open Source 3D Sensor Simulator for Testing Computer Vision Algorithms

John O. Woods and John A. Christian *

West Virginia University, 395 Evansdale Drive, Morgantown, WV 26506, USA
* Correspondence: john.christian@mail.wvu.edu; Tel.: +1-304-293-3263

Academic Editor: Gonzalo Pajares Martinsanz
Received: 27 October 2015; Accepted: 20 January 2016; Published: 29 January 2016

Abstract: 3D sensors such as LIDARs, stereo cameras, time-of-flight cameras, and the Microsoft Kinect are increasingly found in a wide range of applications, including gaming, personal robotics, and space exploration. In some cases, pattern recognition algorithms for processing depth images can be tested using actual sensors observing real-world objects. In many situations, however, it is common to test new algorithms using computer-generated synthetic images, as such simulations tend to be faster, more flexible, and less expensive than hardware tests. Computer generation of images is especially useful for Monte Carlo-type analyses or for situations where obtaining real sensor data for preliminary testing is difficult (e.g., space applications). We present GLIDAR, an OpenGL and GL Shading Language-based sensor simulator, capable of imaging nearly any static three-dimensional model. GLIDAR allows basic object manipulations, or may be connected to a physics simulator for more advanced behaviors. It permits publishing to a TCP socket at high frame-rates or can save to PCD (point cloud data) files. The software is written in C++, and is released under the open source BSD license.

Keywords: 3D sensors; sensor simulators

1. Introduction

Sensors that produce three-dimensional (3D) point clouds of an observed scene are widely available and routinely used in a variety of applications, such as self-driving cars [1], precision agricultural [2], personal robotics [3], gaming [4], and space exploration [5,6]. This widespread use is a direct result of a series of advancements in 3D imaging technologies over the last few decades that has both increased performance and reduced cost.

There are a variety of different types of 3D sensors, each using a different operating principle to generate the 3D point clouds or depth maps. Early systems were primarily based stereovision (comparing images from two cameras separated by a known baseline) [7], but have recently given way to active sensors. Of particular note are time-of-flight cameras [8,9], LIDARs [6], and structured light sensors (as exemplified by the Microsoft Kinect) [4,9].

Falling prices and the increased availability of these 3D imaging technologies have motivated development of such resources as the Point Cloud Library (PCL) [10] as well as new 3D-specific algorithms in the OpenCV library [11,12].

With the proliferation of 3D sensors and the associated pattern recognition algorithms has come a need for easy ways of testing new approaches and techniques. The most straightforward way of obtaining the required test data would be to collect imagery of a real-world object using the specific sensor of choice. This approach comes with a variety of challenges. First, although many 3D sensors are now quite affordable, some of these sensors (e.g., LIDARs) are still too expensive for many research

groups to easily purchase. Second, analyzing the performance of algorithms with data from an actual sensor often requires precise knowledge of the sensor–object pose as well as a detailed 3D model of the observed object. Obtaining such information can be tedious, time consuming, and can significantly impede rapid testing early in the design process. Third, images produced by real sensors have noise and a variety of artifacts associated with their unique operating principles. While the handling of such idiosyncrasies is important for mature algorithms that will be used in practical settings, we frequently want to control the introduction of such effects during algorithm development and testing. Fourth, hardware-based tests are ill-suited for supporting large statistical analyses of algorithm performance, such as in Monte Carlo analyses [13]. Fifth, and finally, some scenarios are extremely difficult to test in the laboratory environment—especially those involving space applications where the scale, lighting, dynamics, and scene surroundings are challenging (and expensive) to accurately emulate on Earth [14–16].

For these reasons it is often preferable to test new algorithms via simulation, only moving to hardware tests when algorithms are more mature. Compared to the hardware-based testing described above, depth images created through computer simulation are cheaper, more flexible, and much faster. Thus, a simulation-based approach is ideal for algorithm development, performance analysis, incremental introduction of error sources, and large Monte Carlo-type analyses in which an algorithm may need to be run tens of thousands of times with randomly varying inputs.

Yet, to our knowledge, no software has been made freely available to the general public for the emulation of a generic 3D sensor—that is, for the easy and rapid generation of simulated depth images from 3D models. While there are good open-source tools for processing and editing point clouds (e.g., MeshLab [17]), these are typically more suited for handling existing point clouds rather than generating the point clouds (or depth maps) that would be created by a 3D sensor. It is also not desirable to generate a measured point cloud directly from the observed model vertices, since we desire the point cloud to describe where the sensor would actually sample the surface rather than simply where the model happens to have mesh vertices. As a result, various research groups are left to independently redevelop such a capability, leading to much duplication of effort and lack of a common tool. We address this problem by presenting GLIDAR (a portmanteau of OpenGL and LIDAR), which is capable of loading 3D models in a variety of formats, re-orienting them, and saving depth images of the visible surfaces (Figure 1). GLIDAR is written in C++ and makes use of the graphics rendering pipeline included in most modern personal computers via the GL Shading Language (GLSL), which confers sufficient speed that it may be used as an input for nearly any algorithm which processes point cloud data.

GLIDAR does not simulate any specific piece of hardware. It can, however, be used in lieu of nearly any 3D sensor and can generate synthetic point clouds of whatever model object the user chooses to load (a few illustrative examples are shown in Figure 2). GLIDAR permits point clouds to be saved as files, visualized, or published to a Transmission Control Protocol (TCP) socket. Additionally, this software—if more complicated dynamics than constant rotation are desired—may optionally subscribe (via TCP) to a physics simulation.

While other groups have utilized graphics hardware to produce RADAR and LIDAR sensor simulators in the past [18–20], GLIDAR appears to be the only piece of software available for free public download. Thus, the purpose of this short note is to introduce the imaging community to GLIDAR and to document key aspects of its present implementation. It is our hope that our software — released under an open source license — will be of use to others, and that users might submit patches and improvements.

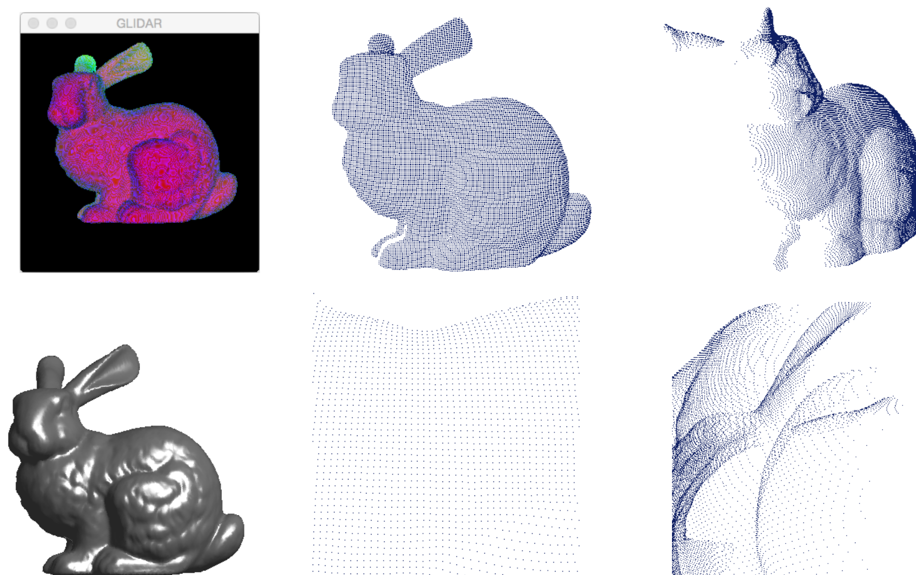

Figure 1. GLIDAR *uses the green and blue channels to store depth information (top left) and outputs a point cloud sampled from the visible surfaces of the object (displayed from a variety of angles and ranges on the right).* Shown at bottom left is a normal 3D rendering of the original model, the Stanford Bunny, which is available from the Stanford 3D Scanning Repository [21].

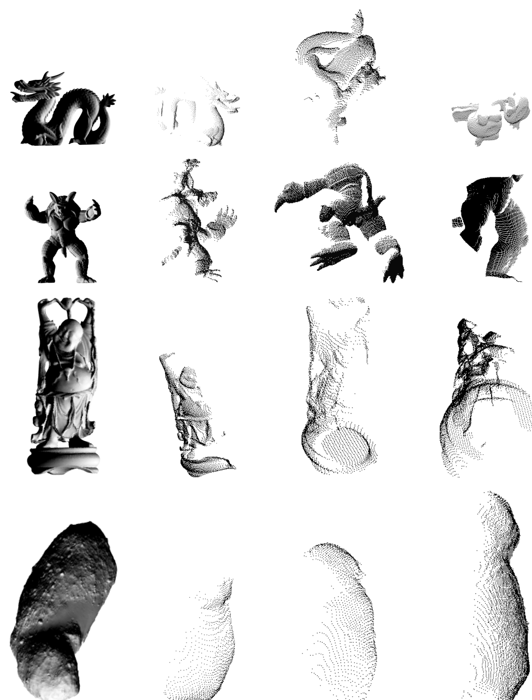

Figure 2. GLIDAR *can quickly produce depth images of objects from different orientations, with differently shaped and sized fields of view, and at a variety of ranges.* The left-most column consists of four models which we used to produce the point clouds in the other columns. The first three models (Dragon, Armadillo, and Happy Buddha) are from the Stanford 3D Scanning Repository [21]. The fourth is the asteroid 25143 Itokawa, and was obtained from then the NASA Planetary Data System [22] and glued together and exported into PLY format using Blender [23].

2. Algorithms

GLIDAR's design is based on a fairly standard pattern for OpenGL applications (which will not be explored in great detail here). In general, the current design does away with color information, utilizing the red channel for intensity and the green and blue channels to store the z depth (the camera frame convention used here assumes that the $+z$ direction is out of the camera and along the camera boresight direction).

We elected not to use the standard depth buffer to store depth information, as this portion of the pipeline was not easily customizable within the graphics architecture on which we developed GLIDAR (a 2011 MacBook Air). The default behavior for the OpenGL depth buffer is to provide the most precision near the camera, and less further away; but we wanted the precision to be equal for points near and far away, and using the color buffer allowed us to directly specify the precision distribution.

Two key modifications to the standard pipeline were necessary: we turned off antialiasing to avoid modifying the depth information stored in the green and blue channels, and enabled double-buffering (it could, perhaps, be disabled if we wished to crudely simulate a scanning LIDAR). This section focuses on the less conventional aspects of the rendering pipeline, namely the GLSL shaders and "unprojection" of the depth information.

2.1. The Programmable Rendering Pipeline

The vertex and fragment shaders, first proposed by [24], are standard parts of the modern rendering pipeline—but are nevertheless worth explaining for the purposes of the discussions that follow.

Primitive objects (points, lines, and polygons) are specified in terms of vertices, but the graphics hardware must provide per-pixel output. Rendering is split into multiple passes, with the least frequent operations (e.g., primitive group) occurring first; the per-primitive group outputs are then processed by the vertex shader in parallel across each vertex. The vertex shader outputs are assembled into primitives for rasterization. Finally, the vertex shader's outputs are interpolated on a per-pixel basis in the fragment shader, where they may be used to compute color, texture, and depth information for each pixel (fragment).

For the purposes of this discussion and for notational convenience, we present the per-vertex operations as if they were global, and per-fragment operations with a subscript i—since the per-fragment computations are performed on interpolated values from the vertex shader. However, it is worth noting that graphics processing units compute each vertex in parallel, and then each fragment in parallel.

In the standard shading models, some basic compromises are made to model the amount of light reflected toward the viewer. For example, [25] shading uses $\vec{r} \cdot \vec{v}$, which must be recalculated for each fragment. In Blinn–Phong shading [26], the expression is $\vec{n} \cdot \vec{h}$, where $\vec{n}$ is the surface normal, and

$$\vec{h} = \frac{\vec{l} + \vec{v}}{||\vec{l} + \vec{v}||} \tag{1}$$

is termed the halfway vector, or simply the half-vector; here, the vector to the viewer is $\vec{v}$, and the light source vector is $\vec{l}$. (The notation $||\vec{x}||$ indicates the 2-norm length of vector $\vec{x}$).

However, when the light source is approximately co-located with the viewer, as in many 3D sensors, we can use the equivalent computation

$$\vec{h} = \frac{\vec{l}}{||\vec{l}||} \tag{2}$$

simply the normalized light vector, for each vertex—a value that is next interpolated into $\vec{h}_i$ for each fragment i.

2.2. Fragment Shader

A typical fragment shader calculates the color and alpha (transparency) of each pixel, or fragment, in an image based on values from the vertex shader. Such a fragment shader would output four values for each fragment: normalized intensities for each of red, green, and blue, and an alpha.

In contrast, the GLIDAR fragment shader outputs only the red value, and uses the green and blue channels to provide the z depth (the length of $\vec{l}$ projected onto boresight direction) interpolated at each fragment; alpha is currently unused (Figure 1).

Attenuation a is calculated for each fragment i using the standard OpenGL lighting model,

$$a_i = \left(a_{\mathrm{C}} + a_{\mathrm{L}} d_i + a_{\mathrm{Q}} d_i^2\right)^{-1} \tag{3}$$

where $d_i = \|\vec{l}_i\|$ is the fragment–sensor (or fragment–light) distance, and a_{C}, a_{L}, and a_{Q} are respectively the constant, linear, and quadratic attenuation values for the light source.

Per-fragment red intensity

$$r_i = a_i c_i \tag{4}$$

is then the product of attenuation and color,

$$c_i = \left(c_{\mathrm{A}} + c_{\mathrm{D}} \max(\vec{n}_i \cdot \vec{h}_i, 0) + c_{\mathrm{S}} \max(\vec{n}_i \cdot \vec{h}_i, 0)^s\right) \tag{5}$$

with $\vec{n}_i$ as the surface normal at the fragment being sampled (interpolated from the vertex normals in the vertex shader), 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* (a term specific to graphics pipeline specifications) is described by s and is used to adjust the relative influence of the specular component in Equation (5). OpenGL allows values of s in the range $[0, 128]$.

We used the green channel to store the most significant bits and the blue to store the remainder. In other words, the green and blue intensities for each fragment i—with each able to hold one byte, and in the range $[0, 1]$—are

$$g_i = \frac{\left\lfloor \frac{D_i}{256} \right\rfloor}{256} \tag{6}$$

$$b_i = \frac{D_i \bmod 256}{256} \tag{7}$$

where D_i is the distance ratio for the fragment; we define

$$D_i = 65536 \frac{\omega_i d_i - n}{f - n} \tag{8}$$

where ω_i is the projection of the light vector direction for the i-th fragment onto the camera boresight direction $\vec{s}$ and is computed according to

$$\omega_i = \frac{\vec{s}}{\|\vec{s}\|} \cdot \frac{\vec{l}_i}{\|\vec{l}_i\|} \tag{9}$$

Thus, the term $\omega_i d_i$ finds the z depth (or the z-axis coordinate) of the i-th fragment as expressed in the camera frame. Again recall that the camera boresight is defined to be along the camera frame's positive z-axis direction. The values n and f in Equation (8) represent the near and far plane z depths,

respectively, in the camera frame. We see, therefore, that our equation for D_i automatically scales the depth value stored for each fragment such that we obtain the best possible resolution given our 16-bit expression of depth (8 bits from the green channel and 8 bits from the blue channel).

2.3. Strategies for Improving Depth Accuracy

In a standard OpenGL depth buffer, greater precision is afforded to points in a scene which are closer to the camera. This default behavior, while useful for 2D rendering, produces point clouds with noticeable depth banding—that is, the depths in a 3D image appear to be quantized when the scene is rendered (see Figure 3). Quantization is reduced, but not eliminated, in two ways:

1. use of both the green and blue channels (as described above) to store depth information, and
2. adjusting the near and far plane locations to hug the object-of-interest as tightly as possible.

As such, the current release of GLIDAR is capable of rendering only one object at a time. Adaptations for rendering entire scenes would require major revisions to our approach for (2).

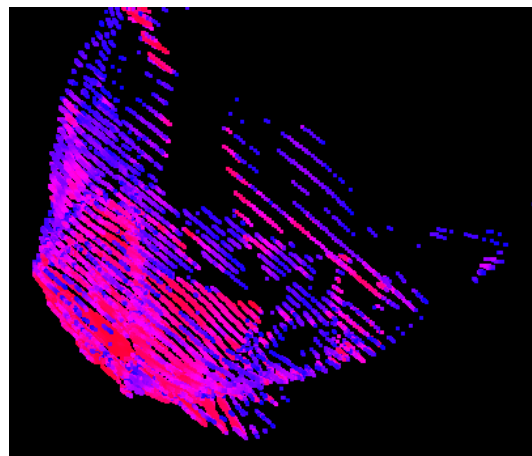

Figure 3. *Depth quantization, in the form of banding, occurs without significant customizations to the storage of depth information.* This point cloud is sampled from a fake rock model, using only the blue channel for depth, and without the other adjustments we made for improving the depth accuracy.

3. Discussion

GLIDAR has been tested on several Ubuntu Linux machines as well as on Mac OS X (both Mavericks and Yosemite). It requires a graphics card with support for GLSL 1.2 or higher.

The software also makes use of a number of open source libraries, and requires CMake, OpenGL Mathematics (GLM) and Eigen, GLEW, GLFW, ØMQ, ASSIMP, ImageMagick/Magick++, and PCL. Some of these requirements, such as Eigen (which is used for matrix and vector operations), are deprecated by our use of GLM, and will be removed in a future release. Additionally, PCL is mainly used for parsing command line arguments (and was used for convenience as part of our own pipeline), but may be dispensed with easily.

Perhaps the most important of the software requirements is the Open Asset Import Library (ASSIMP) [27], which is responsible for loading 3D models in an array of formats—thus, GLIDAR works with most models ASSIMP is able to load. Additionally, our software makes use of Magick++ for loading textures (Figure 4), particularly bump textures, which greatly improve the fidelity of the rendered 3D images by providing normal information for the object models.

In future versions, we hope to develop a texture-based noise model for simulating sensor noise; for a proof of concept generated using GLIDAR, see Figure 5. Two simple noise models are included in the current version, but both are currently disabled in the fragment shader, and neither has been tested extensively.

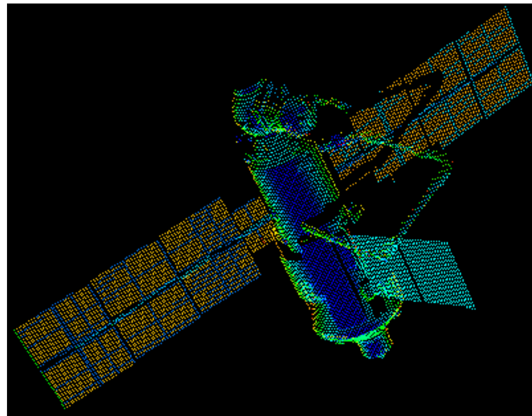

Figure 4. GLIDAR *can apply model texture information to affect the intensity and normals of a model when producing depth images.* In this illustration, the solar panel textures from the International Space Station's Functional Cargo Block module are visible as differing intensities in the point cloud. The texture map used in this example is meant to be illustrative of GLIDAR's capability to consider texture, rather than to represent the actual appearance of such an object; accurate representation of structures in the near-infrared would likely require generation of custom textures. This particular model is a component of NASA's high-resolution ISS scene in Lightwave format, obtained from [28].

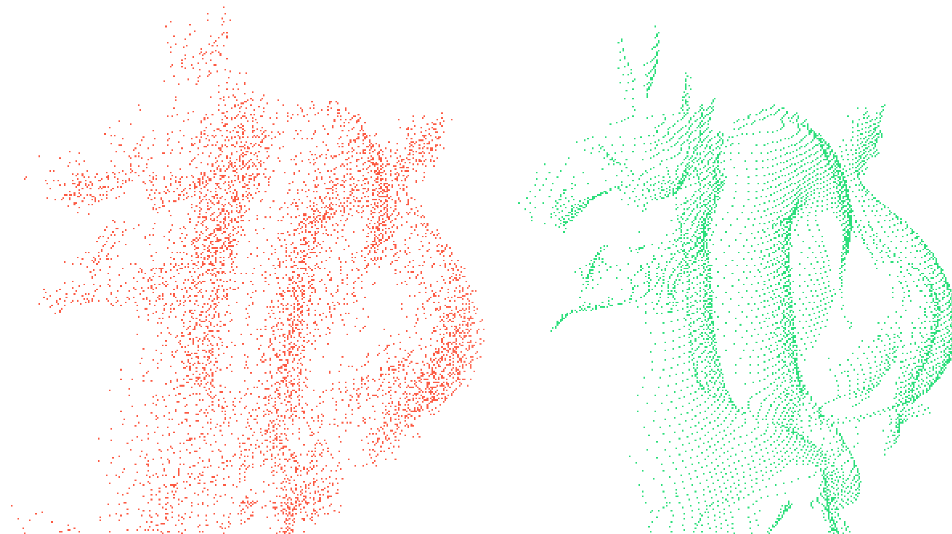

Figure 5. *An additive noise model may be included in the fragment shader to apply device-specific range errors.* Shown above are point clouds sampled from the Stanford dragon, with additive noise (**left**) and without any noise (**right**). We hope to include texture-based noise in the future.

Another area where GLIDAR is not particularly useful is in dealing with deformable models, such as plants, animals, and faces, which are outside the scope of our research team's present needs. However, we encourage others to submit improvements.

Although GLIDAR is not designed to simulate any specific sensor, we find it to be a useful asset for testing and characterizing algorithms that work on point clouds (e.g., iterative closest point, [29]). We hope that future work will further increase the utility of this new software, which may be downloaded from Github at http://github.com/wvu-asel/glidar.

Acknowledgments: The authors wish to thank Thomas Evans, Jordan Sell, and Andrew Rhodes of West Virginia University for many thoughtful conversations and support throughout the preparation of GLIDAR and this manuscript. This research was supported by NASA Goddard Space Flight Center through a contract with the Satellite Servicing Capabilities Office (contract NNG14CR58C, subcontract METSB0043).

Author Contributions: Woods is the primary developer of GLIDAR and authored most of the C++ code that comprises the GLIDAR tool. He wrote Section 2 (Algorithms) and produced the point clouds shown in Figures 1–5. Christian wrote most of Section 1 (Introduction), produced the 3D renderings in Figures 1 and 2, and directed the development activities. Both authors contributed to Section 3 (Discussion).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Levinson, J.; Askeland, J. ; Becker, J. ; Dolson, J. ; Held, D. ; Kammel, S. ; Kolter, J.Z. ; Langer, D. ; Pink, O. ; Pratt, V.; *et al*. Towards Fully Autonomous Driving: Systems and Algorithms. In Proceedings of the IEEE Intelligent Vehicles Symposium (IV), Baden-Baden, Germany, 5–9 June 2011.
2. Hölfe, B. Radiometric Correction of Terrestrial LiDAR Point Cloud Data for Individual Maize Plant Detection. *IEEE Geosci. Remote Sens. Lett.* **2014**, *11*, 94–98.
3. Rusu, R. Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments. *Künstliche Intell.* **2010**, *24*, 345–348.
4. Zhang, Z. Microsoft Kinect sensor and its effect. *IEEE Multimed.* **2012**, *19*, 4–10.
5. Bajracharya, M.; Maimone, M.; Helmick, D. Autonomy for Mars Rovers: Past, Present, and Future. *Computer* **2008**, *41*, 44–50.
6. Christian, J.A.; Cryan, S. A survey of LIDAR technology and its use in spacecraft relative navigation. In Proceedings of the AIAA Guidance, Navigation, and Control Conference, Boston, MA, USA, 19–22 August 2013; pp. 1–7.
7. Kanade, T.; Yoshida, A.; Oda, K.; Kano, H.; Tanaka, M. A stereo machine for video-rate dense depth mapping and its new applications. In Proceedings of the Computer Society Conference on Computer Vision and Pattern Recognition, San Francisco, CA , USA, 18–20 June 1996, pp. 196–202.
8. Foix, S.; Alenyà, G.; Torras, C. Lock-in time-of-flight (ToF) cameras: A survey. *IEEE Sens. J.* **2011**, *11*, 1917–1926.
9. Dal Mutti, C.; Zanuttigh, P.; Cortelazzo, G. *Time-of-Flight Cameras and Microsoft Kinect*; Springer: New York, NY, USA, 2012.
10. Rusu, R.B.; Cousins, S. 3D is here: Point Cloud Library (PCL). In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Shanghai, China, 9–13 May 2011; pp. 1–4.
11. Bradski, G. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*. Available online: http://www.drdobbs.com/open-source/the-opencv-library/184404319 (accessed on 27 October 2015).
12. Bradski, G.; Kaehler, A. *Learning OpenCV: Computer Vision with the OpenCV Library*; O'Reilley Media, Inc.: Sebastopol, CA, USA, 2008.
13. Von Neumann, J.; Ulam, S. Monte Carlo method. *Natl. Bur. Stand. Appl. Math. Ser.* **1951**, *12*, 36.
14. Christian, J.A.; Patangan, M.; Hinkel, H.; Chevray, K.; Brazzel, J. Comparison of Orion Vision Navigation Sensor performance from STS-134 and the Space Operations Simulation Center. In Proceedings of the AIAA Guidance, Navigation, and Control Conference, Minneapolis, MN, USA 13–16 August 2012.
15. Ahlbrandt, S.; Huish, D.; Burr, C.; Hamilton, R. The Space Operations Simulation Center: A 6DOF laboratory for testing relative navigation systems. In Proceedings of the AAS Guidance and Control Conference, National Harbor, MD, USA, 13–17 January 2014; pp. 1–14.
16. Evans, T.; Christian, J.A.; Marani, G.; Lewis, P. Testing Facility for Autonomous Robotics and GNC Systems at West Virginia University. In Proceedings of the AAS Guidance and Control Conference, National Harbor, MD, USA, 13–17 January 2014.
17. Cignono, P.; Callieri, M.; Corsini, M.; Dellepiane, M.; Ganovelli, F.; Ranzuglia, G. MeshLab: An Open-Source Mesh Processing Tool. In Proceedings of the Eurographics Italian Chapter Conference, Salerno, Italy, 2008.
18. Peinecke, N.; Döhler, H.U.; Korn, B.R. Simulation of Imaging Radar Using Graphics Hardware Acceleration. *Proc. SPIE* **2008**, *6957*, 69570L–69570L–12.
19. Peinecke, N.; Lueken, T.; Korn, B.R. Lidar simulation using graphics hardware acceleration. In Proceedings of the 2008 IEEE Digital Avionics Systems Conference, St. Paul, MN, USA, 26–30 October 2008; pp. 1–8.
20. Wang, S.; Heinrich, S.; Wang, M.; Rojas, R. Shader-based sensor simulation for autonomous car testing. In Proceedings of the 2012 15th International IEEE Conference on Intelligent Transportation Systems Anchorage, AK, USA, 16–19 September 2012; pp. 224–229.

21. Stanford 3D Scanning Repository. *Stanford Computer Graphics Laboratory*. Available online: http://graphics.stanford.edu/data/3Dscanrep/ (accessed on 27 October 2015).
22. Gaskell, R.; Saito, J.; Ishiguro, M.; Kubota, T.; Hashimoto, T.; Hirata, N.; Abe, S.; Barnouin-Jha, O.; Scheeres, D. Gaskell Itokawa Shape Model V1.0. HAY-A-AMICA-5-ITOKAWASHAPE-V1.0. NASA Planetary Data System, 2008. Available online: http://sbn.psi.edu/pds/resource/itokawashape.html (accessed on 29 January 2016).
23. Blender. Available online: http://www.blender.org (accessed on 27 October 2015).
24. Proudfoot, K.; Mark, W.R.; Tzvetkov, S.; Hanrahan, P. A real-time procedural shading system for programmable graphics hardware. In Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), Los Angeles, CA, USA, 12–17 August 2001, pp. 159–170.
25. Phong, B.T. Illumination for Computer Generated Pictures. *Commun. ACM* **1975**, *18*, 311–317.
26. Blinn, J.F. Models of Light Reflection for Computer Synthesized Pictures. In Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques, New York, NY, USA, July 1977; pp. 192–198.
27. ASSIMP: Open Asset Import Library. Available online: http://www.assimp.org (accessed on 27 October 2015).
28. ISS (High Res). NASA 3D Resources Available online: http://nasa3d.arc.nasa.gov/detail/iss-hi-res (accessed on 27 October 2015).
29. Besl, P.J.; McKay, N.D. A method for registration of 3-D shapes. *IEEE Trans. Pattern Anal. Mach. Intell.* **1992**, *14*, 239–256 .

© 2016 by the authors; licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons by Attribution (CC-BY) license (http://creativecommons.org/licenses/by/4.0/).