

6-12-2013

Kosmos : a virtual 3-D universe

John Judnich
Santa Clara University

Follow this and additional works at: https://scholarcommons.scu.edu/cseng_senior



Part of the [Computer Engineering Commons](#)

Recommended Citation

Judnich, John, "Kosmos : a virtual 3-D universe" (2013). *Computer Engineering Senior Theses*. 3.
https://scholarcommons.scu.edu/cseng_senior/3

This Thesis is brought to you for free and open access by the Engineering Senior Theses at Scholar Commons. It has been accepted for inclusion in Computer Engineering Senior Theses by an authorized administrator of Scholar Commons. For more information, please contact rscroggin@scu.edu.

Santa Clara University
DEPARTMENT of COMPUTER ENGINEERING

Date: June 12, 2013

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY
SUPERVISION BY

John Judnich

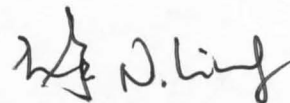
ENTITLED

Kosmos: A Virtual 3-D Universe

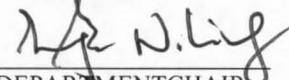
BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING



THESIS ADVISOR



DEPARTMENT CHAIR

KOSMOS: A VIRTUAL 3-D UNIVERSE

by

John Judnich

SENIOR DESIGN PROJECT REPORT

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science in Computer Engineering
School of Engineering
Santa Clara University

Santa Clara, California

June 12, 2013

ABSTRACT

Kosmos is an application enabling interactive visualization of a fictional 3D universe. It offers users the opportunity to explore and experience an aesthetically pleasing virtual environment complete with billions of high-resolution planets and stars. Kosmos integrates several novel 3D rendering techniques in terrain rendering, large-scale particle systems, etc. to make this level of graphical realism and scale possible. The efficiency of the algorithms developed for this project enables average hardware (such as almost any modern laptop) to run Kosmos smoothly. Moreover, through the use of the recent WebGL standard, Kosmos may be viewed online¹ in any modern web browser on any major operating system, with no large downloads or additional software installation necessary. Moreover, the author has released the full source code for Kosmos online for free under the BSD Open Source License².

¹ Kosmos may be viewed with at judnich.github.io/Kosmos/ (note: may not run on older computers)

² Full source code for Kosmos (plus additional documentation) is available at github.com/judnich/Kosmos

ACKNOWLEDGEMENTS

The author would like to thank the Dr. Nam Ling and the School of Engineering for continued help towards providing research work, funding, and financial aid, without which the author's undergraduate education would not have been financially possible.

LIST OF FIGURES

Figure 1. "Moon" screenshot.....	11
Figure 2. "Alien sunset" screenshot.	11
Figure 3. Illustration of cube to sphere mapping used for planets.	22
Figure 4. Enumeration of the 23 symmetric cluster sets used in SCSLOD.	22
Figure 5. Cross-sectional illustration of a depth interval grid.....	23
Figure 6. Illustration of Sharp Normal Map nonlinear interpolation.	25
Figure 7. Comparison of regular normal maps (top) vs. Sharp Normal Maps (bottom)...	26
Figure 9. Kosmos introduction screen.....	35

LIST OF TABLES

Table 1: Functional Requirements	13
Table 2: Nonfunctional Requirements	14
Table 3: Project Risks	15

TABLE OF CONTENTS

INTRODUCTION	9
DESIGN AND IMPLEMENTATION STRATEGY	12
FUNCTIONAL REQUIREMENTS.....	13
NONFUNCTIONAL REQUIREMENTS.....	14
TECHNOLOGIES USED.....	14
DEVELOPMENT TIMELINE.....	14
PROJECT RISKS.....	15
TEST STRATEGY AND USE CASES.....	16
SYSTEM ARCHITECTURE	17
QUERY BASED RENDERING.....	17
CONTENT QUERY CACHE WITH CONCURRENT LOADING.....	18
COMPUTATION ON THE GPU.....	19
IMPLEMENTATION DETAILS	21
PLANET TERRAIN RENDERING.....	21
HIGH RESOLUTION NEAR SURFACE RENDERING.....	23
“SHARP NORMAL MAPS”.....	24
FAST FRUSTUM CULLING.....	27
GENERAL PROCEDURAL GENERATION NOTES.....	27
STARFIELD GENERATION AND RENDERING ON THE GPU.....	28
STAR RENDERING / SHADING.....	29
CONCURRENT GENERATION OF HIGH-RESOLUTION PLANET DATASETS ON THE GPU ...	30

SOLVING PRECISION LIMITATIONS.....	32
EMULATED 128 BIT COORDINATES	32
USING FLOATING ORIGIN ON THE GPU	32
Z-BUFFER PRECISION CONCERNS AND MITIGATION: LAYERED DEPTH RENDERING	33
USER INTERFACE.....	35
INTRODUCTION SCREEN.....	35
AUTOPILOT BUTTON: “INTELLIGENTLY” AIDED VIEW CONTROLS	35
SHARE BUTTON	36
LOCATION PERSISTENCE.....	37
BROWSER COMPATIBILITY WARNING	37
DATA FILE FORMAT	37
SOCIETAL ISSUES	39
OPEN-SOURCE DECISION.....	39
CONCLUSION.....	41
LESSONS LEARNED	41
FUTURE PLANS	43
REFERENCES.....	44
APPENDIX	45

INTRODUCTION

“Kosmos” is a 3-D visualization of a fictional universe containing billions of stars and planets that runs on relatively inexpensive computing devices. Moreover, Kosmos is widely accessible simply through a modern web browser (with no large downloads or software installation required). This visualization engages the users by presenting an interaction mechanic that leads the user continuously through a wide range of view scopes within the galaxy. Kosmos enables the user to obtain some sense of the massive scope of planets, stars, star clusters, etc. as they fly around and interactively explore this virtual universe from many perspectives and viewpoints. The full source code for Kosmos is available to the public for free under the BSD Open Source Licence [1].

Kosmos represents an intersection of academic research in algorithms, software engineering implementation, and art. The development of Kosmos has involved:

- Cutting edge academic research towards pushing the boundaries of modern computer technology, particularly (in this case) 3D rendering algorithms and mobile graphics hardware.
- A reasonably large engineering effort spanning multiple implementation iterations while technology is incrementally improved, new algorithms created, and better engineering practices learned.
- Artistic expression, and an interpretation of the natural beauty of the massive scales and structures in the real universe.

In other words, Kosmos targets three seemingly separate but co-dependent problems:

1. Mobile 3D graphics technology needs to be improved to enable progressively richer, higher resolution, and larger scale visualizations on mobile devices.
2. There are no 3D libraries currently available that include support for the visualization of large 3D planets, billions of stars, etc.
3. There are no mobile or browser-based interactive visualizations of a virtual 3D universe that allow you to freely explore a vast area of space at resolution high enough to see details on planet surfaces.

Problem #1 is a research problem, which stands on its own as a valuable area to explore and push the boundaries of technology. The majority of time for this project has been devoted to such research, resulting in several successful contributions to the field of 3D rendering algorithms (including some beneficial even beyond the scope of this particular project) including a full technical paper currently under review by the journal IEEE Transactions on Visualizations and Graphics, plus conference papers already published.

Problems #2 and #3 above are market problems. Upon further research since this project's initial proposal, the author discovered full-scale universe visualization software does already exist³. Rather than continuing in a similar direction to this existing software, the author chose to retarget this project towards a different demographic.

Existing Software ("Space Engine") [2]

- Requires a large >500 MB download
- Requires installation on the computer's hard drive
- Requires Windows (no Mac, Linux, or mobile support)
- Requires a powerful CPU and GPU found only in expensive computers

Our Software ("Kosmos")

- Runs on an average modern laptop (and eventually mobile devices, e.g. tablets)
- Requires only a modern web browser – no additional installation necessary
- No large downloads – the web application loads almost immediately

In particular, the latest version of Kosmos is designed to run on the average modern laptop. Most significantly, no software installation or large downloads are necessary because the entire application will be available *directly* from any modern web browser (facilitated by the recent web standard technology "WebGL"). It is not necessary to download large amounts of data since most of the content is generated mathematically in real-time, and thus the entire application compressed to an extremely small overall size.

Figures 1-2 show some screenshots produced directly by Kosmos, unmodified.

³ en.spaceengine.org

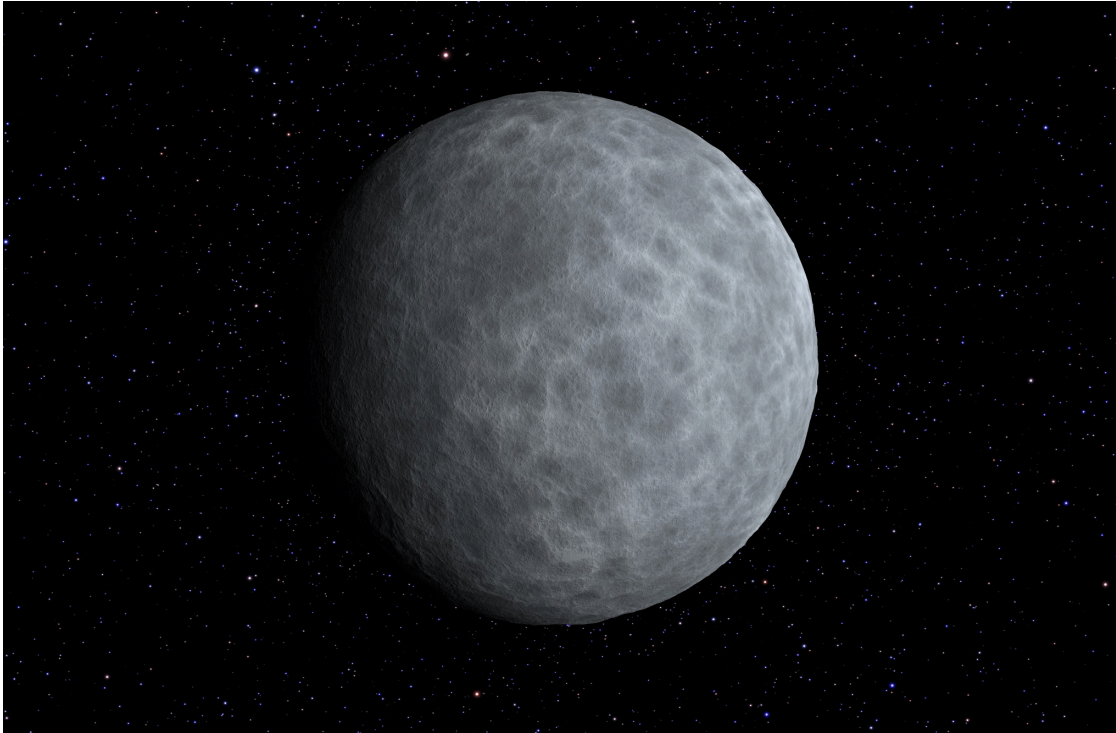


Figure 1. "Moon" screenshot.

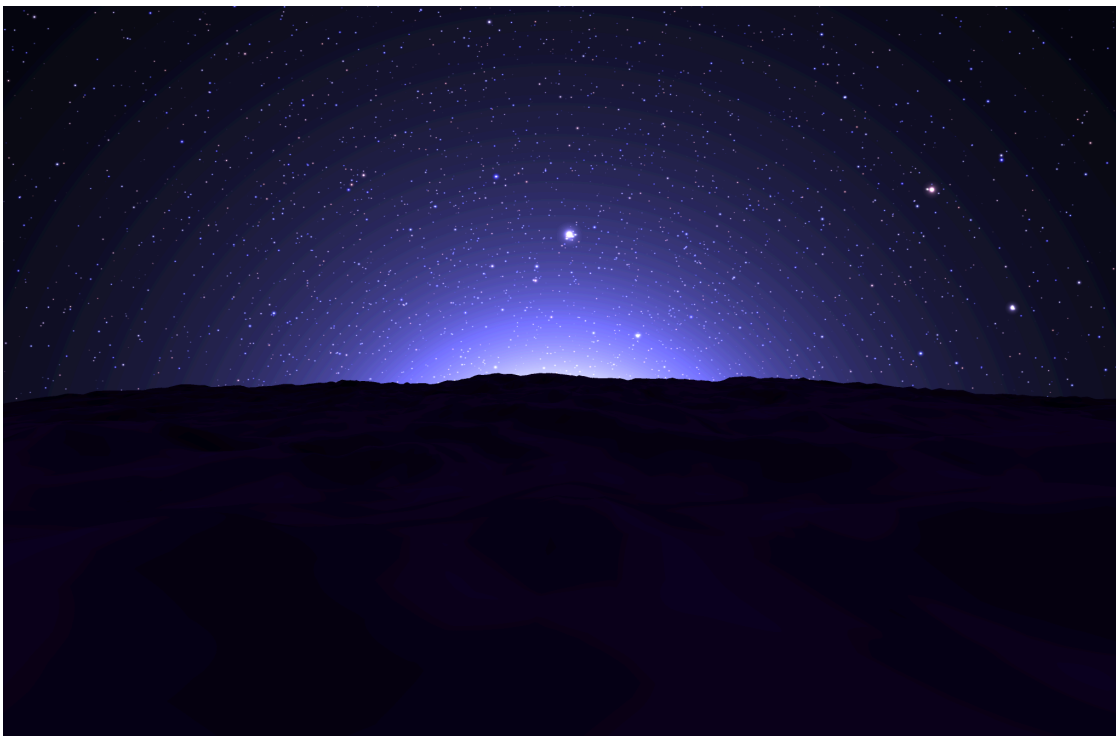


Figure 2. "Alien sunset" screenshot.

DESIGN AND IMPLEMENTATION STRATEGY

The overall design philosophy for Kosmos is simple:

- *Value content quality over quantity*
- *Value understated simplicity over impressive complexity*

These principles are aimed at ensuring the end product provides a consistently high quality user experience for our target audience.

The implementation strategy consists of two major guidelines:

- *Iterative bottom-up development with frequent refactoring*
- *Composition or component oriented design*

Composition/component oriented design refers to minimizing the abuse of object oriented programming features generally considered harmful (e.g. bloated inheritance trees).

Iterative development is an extremely effective approach to rapidly developing high quality, complex software. This method of software development resembles a process of iterative evolution, as opposed to older (and usually failed) efforts to design entire software system architectures before prototyping or implementing them.

For example, this project began initially as a native application using C++ and OpenGL, only running on powerful NVIDIA video cards. Although these early prototypes provided valuable experience and research material, in the end all of these iterations were discarded in favor of a different target platform (this was due to the discovery of SpaceEngine [2], which already accomplished goals similar to the original Kosmos project).

Due to lessons learned from these previous prototypes, the latest web-based version is much more efficient with an overall vastly improved software architecture. Most importantly, the iterative development model allows for rapid reaction to market changes as demonstrated by the successful late rewrite of this project. This kind of response is not otherwise possible with more rigid and antiquated processes (e.g. waterfall).

Functional Requirements

Functional requirements represent the core set of features that must be implemented to complete the project. These requirements are listed below.

P1 features are required. P2 and P3 are optional, with different levels of priority.

Table 1: Functional Requirements

Priority	Requirement Name	Requirement Description
P1	Planet Terrain	The graphics engine shall display 3D planets with detailed high-resolution mountains and ground level features.
P3	Atmospheric Scattering	The graphics engine should simulate 3D atmospheric scattering to display an atmospheric glow from space and realistic sky coloring from within the atmosphere.
P1	Stars	The graphics engine shall visually represent millions of stars in 3D space, each of which may have orbiting planets.
P3	Nebulae	The engine may visually represent gas clouds and nebulae within the virtual 3D universe.
P2	Exponential View Speed Control	The user interface should provide the ability to modify viewer speed on an exponential scale.
P2	Automated Speed Control	The user interface should provide a mode (enabled by default), which automatically scales the viewer's speed based on proximity to planetary bodies to ease the user's navigational task.
P3	Automated Viewer "Upright" Orientation	The engine should automatically orient the viewer upright when approaching planets (since this is the only context where "upright" is properly defined).

Nonfunctional Requirements

Nonfunctional requirements are aimed at ensuring the end product provides a high quality user experience for Kosmos’s target audience. These requirements are listed below.

Table 2: Nonfunctional Requirements

Constraint Description	Optimization Strategy
Performance maximization	Routinely benchmark and profile whole program performance. Research, develop, and implement the most efficient algorithms for each computationally demanding feature.
Aesthetic visual quality refinement	Routinely obtain and review subjective feedback from random samplings of the program’s target audience.
Intuitive visual interface	(Same as above)

Technologies Used

- CPU Languages
 - CoffeeScript, JavaScript
- GPU Language
 - GLSL
- APIs
 - WebGL (JavaScript OpenGL bindings supported on modern browsers)
 - jQuery (JavaScript cross-browser compatibility library)
 - glMatrix (JavaScript library providing simple matrix math operations)

Development Timeline

The latest web-based version of Kosmos was rewritten completely from scratch in about six weeks, with around two weeks of break in the middle. Therefore a full Gantt Chart does not make sense for this project (since too many features were completed at too fast of a pace to document individually). Note that a Gantt Chart could be used to illustrate

the development of prior prototypes (which occurred over a longer period), but these are not really relevant to the current version of Kosmos, since they use a different set of technologies and techniques.

Project Risks

Table 3: Project Risks

Risks	Consequences	P	S	I	Mitigation Strategy
Conflicting schedule	Group is unable to meet for decision-making	0.5	1	1.00	Default project leader can make decisions in the event of schedule failure. Schedule important meetings ahead of time.
Insufficient time to satisfy formal requirements	Desired features may not be implemented	0.5	3	1.50	Prioritize features in order of desired importance
Data loss	Restart project	.01	10	0.10	Heavily redundant backup including multiple server-based and physical copies.
Insufficient application performance	End users will have higher system requirements than desired	0.8	4	3.20	Apply the best algorithms from cutting-edge research to computationally expensive processes.
Unsatisfactory aesthetic quality	End users do not find visual experience pleasing.	0.5	5	2.50	Gather and respond to user feedback frequently through development.
Unsatisfactory user interface quality	End users will have unpleasant experience interacting with software.	0.2	5	2.50	Gather and respond to user feedback frequently through development.

Test Strategy and Use Cases

Kosmos testing strategy:

- Unit test self-contained components which have clearly defined behavior
- Use automated regression testing for unit tests and performance metrics
- Manually test subjective visual elements
- Manually test complex nondeterministic simulation interactions

Due to nondeterministic user input tied to the simulation path, the correct results of a complex interactive simulation cannot be predetermined to the formal precision necessary of automated unit testing. Moreover, subjective aspects like visual aesthetic quality have no known formal metric. Thus, some areas must be tested manually, such as:

- Interaction dynamics and logic
- User engagement and entertainment factor
- Overall look and feel, user experience
- Aesthetic quality of visual content

From the end-user perspective, the following use cases must be tested as well:

- User opens web application and reads instructions
- User navigates around using the control scheme
- User closes application and resumes later at the same location

SYSTEM ARCHITECTURE

Since Kosmos is an interactive visualization/simulation, discrete state diagrams and flow charts are extremely simple. Usually there is only one state, and a flow chart consists of a simple infinite loop with termination only when the application ends.

Due to power saving features however, Kosmos uses a simple two-state simulation:

- **Active State:** If the user's view remains a static unmoving image and no content is being concurrently loaded, change to idle state; else, repeat the active state.
- **Idle State:** If the user clicks on any control, forward the data to the appropriate function and change to active state. Else, remain idle, conserving power.

Beyond this extremely simple high-level state, the majority of complexity for simulations usually occurs in a more “continuous” state of the application, represented by a large universe of data, data structures, and viewer coordinates. Each will be described in moderate detail below in the appropriate sections.

Query Based Rendering

The most important architectural design of Kosmos is how 3D content from such a massive universe is managed and rendered seamlessly to the screen without pausing at any time to load additional content. In this paper, the architectural design addressing such requirements is called “Query Based Rendering.” In contrast to traditional architectures for rendering large 3D worlds, Query Based Rendering is nearly stateless and allows the render process to be completely decoupled from content management and loading.

Rather than maintaining progressively updated data structures through the course of the user's movement through the world, Query Based Rendering emphasizes using decoupled database(s) specifically optimized for spatial queries (requests for content given a location and radius, for example). This is in contrast to traditional 3D rendering approaches for large worlds, where large, complex, and confusing data structures are maintained manually to track nearby objects as the viewer moves through the world.

While Query Based Rendering is not an entirely new idea (for example, older 2D game engines often used similar approaches), its application as a unified architectural rule to 3D rendering engines seems to be unique.

Query Based Rendering accommodates LOD (level-of-detail) optimizations extremely well, since resolution can easily be made a parameter to the spatial query. For example, the render process may issue several spatial queries covering concentric rings (different radius intervals from the viewer), requesting a different resolution at each distance. Then, the render process can pass the query results to draw functions, achieving a clean and consistent way to implement a progressive LOD.

Note that Query Based Rendering cannot be entirely stateless, since that would imply everything on the screen passes through query calls every single frame (which would be very inefficient, and instantly bottleneck CPU-GPU bandwidth). For this reason, this architecture assumes the database will be custom designed for extremely fast queries, with a VRAM (video memory) content caching scheme to efficiently deal with content that takes considerable time to load. Therefore from the render process's view, all of this caching behavior can be ignored because it's already taken care of. As a result, the render process does not need to manage any loading/unloading of content, but simply focus solely on drawing content to the screen. This decoupling is a big advantage over traditional architectures, because the complexity of Kosmos's render system would otherwise be nearly unmanageable without it.

Content Query Cache with Concurrent Loading

Note: This particular implementation of this content query cache is specific to Kosmos; not all 3D rendering engines necessarily need to use the same approach here in order to fall under the "Query Based Rendering" design theme.

Kosmos implements a generic caching mechanism that can be applied to any persistent content in the game world, as long as a unique identifier can be produced for that content. At a high level it operates as an ordinary cache. You instantiate a ContentCache object, providing a callback function that loads and returns content for a given contentId. Then,

whenever you need some content, you request it through the ContentCache's `getContent()` function. This function will return the content immediately if it is already loaded (stored in an internal id-content hash map), or forward the request to the callback function otherwise.

In addition to this standard caching behavior however, Kosmos's ContentCache class supports progressive loading – it allows the callback loader function to load only a fraction of the entire resource in a given call, if desired. This feature is integrated by returning a pair value from the loader callback (percent complete, and intermediate content). If such a pair is returned, the ContentCache intelligently puts it back into the loading queue. Then, the next time ContentCache's `update()` method is called, the load callback is issued again. This process repeats indefinitely until the load callback finishes loading the entire resource, at which point it will return a 100% loaded value and the ContentCache will move it to the `jobCompleted` queue.

This system provides a clean and unified interface for concurrently loading complex and large media files, without disrupting the natural cache interface. The only accommodation required for this asynchronous loading behavior is of course the realization by users of a ContentCache that the `getContent()` method may return null for perfectly valid resources until they finish loading in the background.

For a more technically detailed specification of Kosmos's ContentCache, simply refer to the source code (since an English or diagram based specification takes far more space yet is still less precise).

Computation on the GPU

Since Kosmos is implemented as a web app, the only CPU programming language currently possible is JavaScript. Unfortunately while modern browsers run JavaScript impressively fast for a scripting language, it's still considerably slower than native compiled code. Since efficiency is an important goal of Kosmos, this performance issue presents a technical challenge. The design of Kosmos approaches this issue by emphasizing offloading as much work as possible to the GPU (using GLSL programs via

WebGL). GPU-based computation makes decent performance possible, since GLSL is executed at native speed.

IMPLEMENTATION DETAILS

The algorithms and architectural designs incorporated into the project are overviewed briefly below. An extensive description of every technical component is not possible in a relatively short design project thesis, since each section below could be expanded into over 10 pages of precise technical detail.

However, particularly promising algorithms and tools developed may branch off into their own papers and open-source projects where each technology is documented in-depth. For example, Tuple Markup Language has branched off into its own open-source project with individual in-depth documentation (see github.com/judnich/TupleMarkup). Also, several technologies outlined below are published or under review for publication as academic papers.

Finally, note that the most precise design specification possible is simply the source code itself, which is provided in full 100% free and open-source (released under the BSD Open Source License) at: <https://github.com/judnich/Kosmos>

Planet Terrain Rendering

Relatively undistorted spherical tessellation is achieved with a cube whose six faces are regular grids, whose vertices are then mapped to a sphere using simple vector normalization (see Figure 3).

Each cube face is then rendered as a quad tree of grid nodes using the “Symmetric Cluster Set Level of Detail” terrain rendering algorithm. For more information on this technique, see our conference papers [3][4]. The abstract for the most recent paper follows:

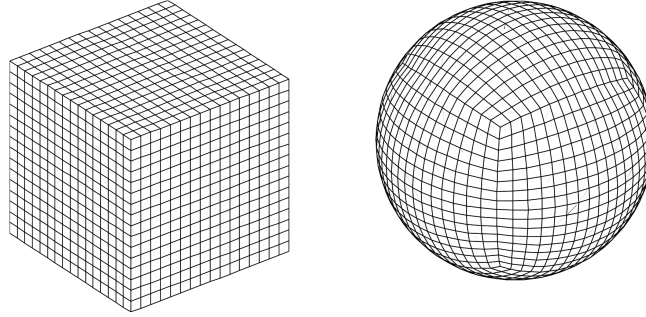


Figure 3. Illustration of cube to sphere mapping used for planets.

Abstract: In this paper, we present an improvement for batch-based quadtree terrain rendering that drastically reduces the number of draw calls to the graphics processing unit. As a result, more fine-grained triangular optimization is possible without sacrificing triangle throughput. No extra preprocessing is required. In general, quadtree terrain algorithms recursively subdivide mesh geometry to meet visual error constraints. Batch-based techniques use buffered grid blocks as the subdivision primitive for better triangle throughput [5]. We base our algorithm on structural observations of such terrain quadtrees. First, we show that the four sub-nodes of any non-leaf can be categorized into sixteen distinct states of drawing behavior. These states are symmetric in such a way that allows just five unique geometries to represent all of them. With the additional observation that leaf nodes appear in groups of four across regions of homogeneous grid resolution, we develop a technique employing 23 unique geometric batches from which any terrain can be rendered (see Figure 4). The resulting algorithm reliably reduces draw calls by a factor of 6 on average, and achieves render performance 30 to 50 percent faster than comparable techniques [6].

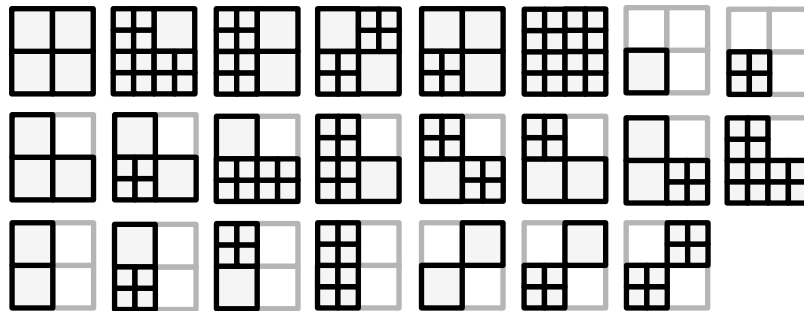


Figure 4. Enumeration of the 23 symmetric cluster sets used in SCSLOD.

High Resolution Near Surface Rendering

For more information on this technique, see our technical paper draft [7], the abstract for which follows:

Abstract: In this paper, we present a novel space-skipping algorithm for per-pixel displacement map rendering on the GPU that significantly accelerates ray convergence. Without any preprocessing, this technique achieves raw displacement map rendering performance two to four times as fast as other techniques without loss of visual quality. Displacement mapping algorithms accurately render complex shapes projected within flat geometry by casting rays from the viewer into a tangent-space height-field; this enables otherwise unachievable resolutions of 3-D detail in games and simulations [8]. Our algorithm accelerates ray convergence by first computing ray boundary intervals within screen-space uniform pixel blocks (see Figure 5). The final pass then uses the dynamically computed depth intervals to converge on a precise intersection point very rapidly. As a result, our algorithm enables rendering of raw displacement map data significantly faster than existing algorithms. Moreover, this technique can provide performance improvement to other existing per-pixel displacement mapping algorithms as well, with or without preprocessing.

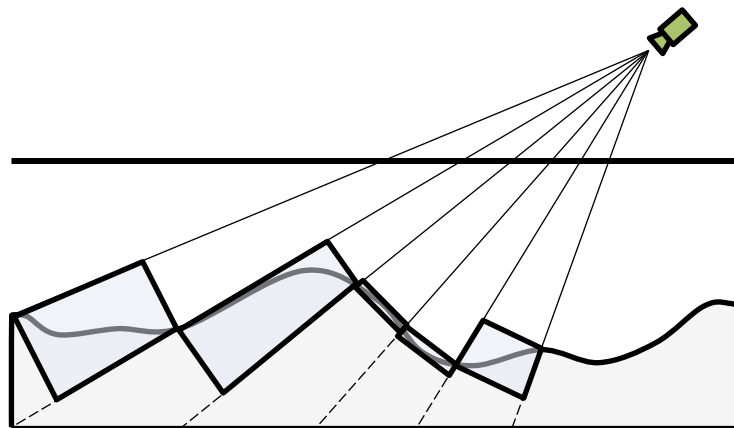


Figure 5. Cross-sectional illustration of a depth interval grid.

“Sharp Normal Maps”

Most 3D rendering engines use lighting models (the mathematical model that calculates the color of pixels on the screen) that make use of surface normal vectors describing surface features. It is therefore extremely important to store high-resolution normal maps for best results. As a result, all modern graphics processing units support built-in hardware support for simple fixed-rate image compression to enable higher resolution normal maps under the same memory constraints (e.g. DXT texture compression ubiquitously available from the OpenGL API).

Sharp Normal Maps introduces a new resolution-enhancing method designed specifically for normal map data, which has the potential to reduce normal map size by a factor of four without loss of visual quality. This technique works *in addition* to existing DXT compression, thus multiplying even further the feasible normal map resolution under given memory constraints.

In practice, normal maps tend to contain a broad distribution of surface feature frequency (low frequency shape, combined with occasional high frequency regions producing “sharp” features). This is especially true of terrain shapes with ridges and valleys found in Kosmos: valleys consist of relatively low frequency features, while peaks and ridges are quite small and require high frequency detail to appear correctly. Sharp Normal Mapping takes advantage of this structural feature of normal maps through an unconventional use of the hardware-accelerated linear interpolation behavior used in computer graphics.

From an information-theoretic perspective, three-dimensional unit-length surface normal vectors are wasteful since the surface of a unit sphere is two-dimensional. Sharp Normal Maps use this third dimension of information to store a “sharpness factor”. Specifically, Sharp Normal Maps divide the length of normal vectors by this “sharpness factor”.

Understanding why downscaling the normal vectors actually affects the interpolation behavior of the per-fragment vectors requires understanding of how the GPU interpolates between texture/vertex attributes. Intuitively, one would want normal vectors to be spherically interpolated (that is, the angle between two vectors interpolated linearly).

GPU hardware however is generic and does not change interpolation behavior for different data. All data is linearly interpolated; when supersampling (magnifying) a normal map, simple linear interpolation is used between each discrete normal vector sample. As a result, interpolated normal values for intermediate locations are not unit vectors, even with traditional normal mapping. As expected, modern 3D engines renormalize the vector values on a per-fragment basis (the fragment shader simply rescales the vector to normal length) to resolve this problem.

Therefore, by scaling the relative length of normal vector samples, although the renormalized unit normal vectors at the endpoints are identical, the linear interpolation in between behaves differently. In particular, depending on the source/source vector length ratio, the renormalized normal vectors will be interpolated with a function ranging anywhere from the regular linear behavior, to extremely nonlinear “accelerated” interpolation. This behavior is roughly illustrated in Figure 6.



Figure 6. Illustration of Sharp Normal Map nonlinear interpolation.

This technique actually creates a look that is impossible to replicate fully by even extremely high resolution normal maps without Sharp Normal Mapping, but for simplicity one can mathematically examine only the intermediate interpolation midpoints and modify appropriately with correctly chosen “sharpness factors” at each point to emulate normal maps twice the horizontal and vertical resolution. The mathematics for computing the corresponding “sharpness factor” to produce a particularly higher resolution output is straightforward but messy (involving a complex arctangent ratio equality). See Figure 7 for a visual comparison of Sharp Normal Maps vs regular normal maps using exactly the same resolution for both.

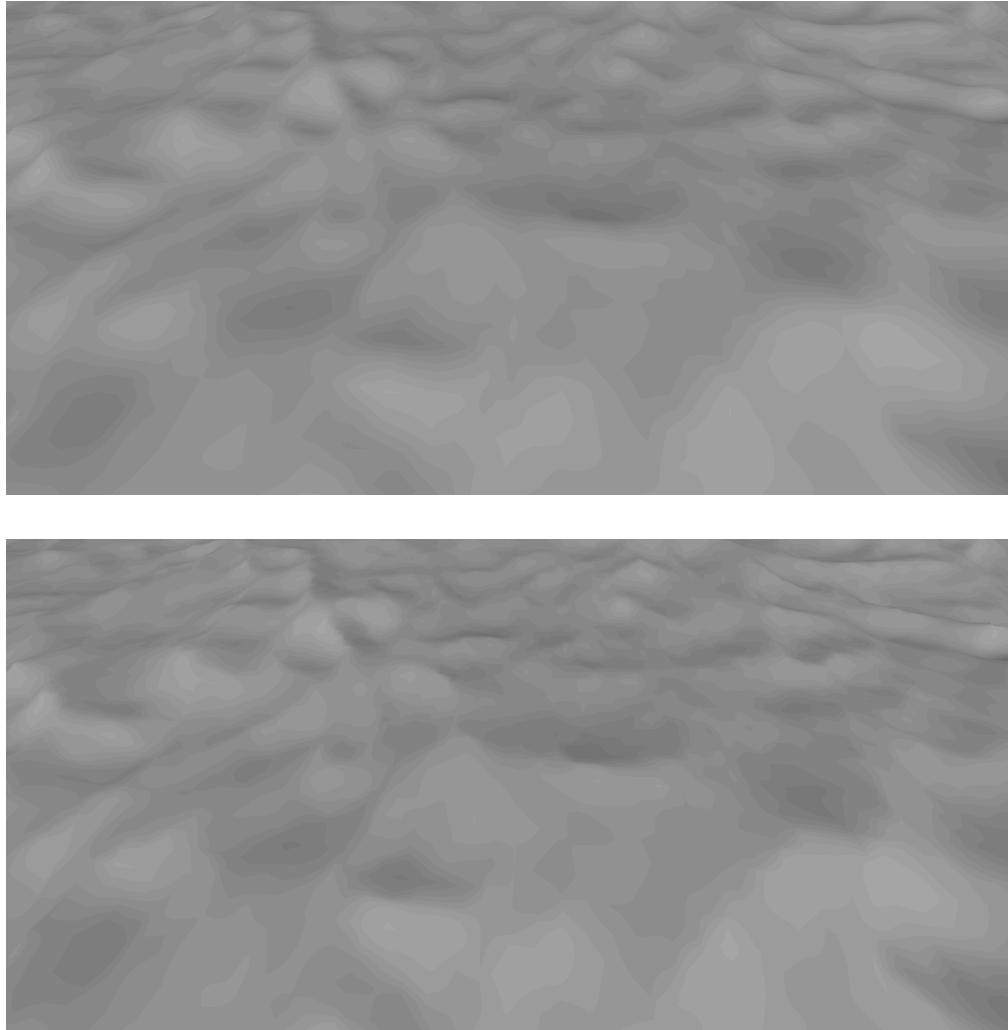


Figure 7. Comparison of regular normal maps (top) vs. Sharp Normal Maps (bottom).

Notice that any 3D rendering engine that uses normal maps (i.e. almost every 3D game engine in operation today) can immediately make use of Sharp Normal Maps to either achieve up to 4x reduction in memory usage or a 2x2 normal map resolution improvement. Moreover, since the per-fragment renormalization is already being performed, Sharp Normal Maps incurs literally *zero* performance or memory overhead.

Research has so far revealed that Sharp Normal Maps appears to be a completely new invention, despite its extreme implementation simplicity. A technical paper and open-source normal map compression tool is being written to enable the rest of the industry to use this technique.

Fast Frustum Culling

Existing frustum culling techniques today usually perform a bounding sphere check followed by a more precise bounding box check. The bounding sphere check is faster, preventing the slower bounding box check from wasting CPU time from obviously occluded objects.

Kosmos presents a better frustum culling technique using dual bounding spheres: one sphere represents the max distance of all vertices from an object's origin; another represents the min distance of all vertices. During frustum checks, an object's point-plane distance is checked against one of the frustum planes, and simultaneously checked against the min and max. This technique effectively performs early culling for almost all objects both inside and outside of the viewing frustum.

The only objects indeterminate by the dual-sphere check are objects likely directly intersecting a frustum plane. This reduces the number of frustum culling candidates from $O(n^3)$ to $O(n^2)$. As a result, the indeterminate objects may check full bounding convex hulls against the frustum plane without concern for performance. This method results in more efficient use of both CPU and GPU compute resources.

General Procedural Generation Notes

All content within Kosmos's virtual universe is procedurally generated through a combination of pseudorandom number generators and various mathematical filters. This means everything you see in Kosmos is completely computer/mathematically generated; Kosmos does not use any external media/resource files of any kind.

For example, planet terrain heightmaps are generated from a spatially parameterized multifractal combining smooth perlin noise, ridged noise, etc. into a fractal dataset that resembles a combination of mountains, erosion, rolling hills, etc.

Procedural generation is custom-designed for each type of content (terrain, star cluster formation, etc). Computer-generated content can be selectively overridden with "real world" datasets if desired, although right now this is not done.

Since the universe contains billions of stars and planets, it makes no sense to pre-generate and save their data. As the viewer's position moves and necessitates more data to render various details, it is generated on-demand in real time.

Starfield Generation and Rendering on the GPU

Draw calls to the GPU are associated with considerable overhead due to driver and PCI bus latency; too many draw calls will leave the GPU idling, waiting for commands and doing nothing most of the time. Therefore, a common optimization when rendering large numbers of simple objects is known as “batching”, where many objects are merged into large data buffers in memory and drawn with a single command to the GPU.

Kosmos renders stars as a rolling 3D grid of cubic star batches. Rendering stars queries the nearest 9 cube blocks surrounding the viewer, culling out blocks that are outside of the field of view. Then each block is rendered with a single draw call. Each block contains several thousand stars generally, and represents the stars within that region of space. The intensity of stars near the maximum view range cutoff is distance-modulated from the vertex shader to create a gradual fade-out effect.

Note that due to JavaScript (as well as general) performance issues, populating new batches of stars dynamically could incur serious lag as the viewer moves rapidly through space. This is undesirable, since the ability to move at any speed smoothly is a goal of Kosmos.

To solve this technical challenge, Kosmos uses an unusual method to generate pseudorandom star positions on the GPU. This works by generating a large vertex buffer containing around 10,000 randomized stars pre-batched and ready to render when Kosmos first loads. Then, whenever a cube block is being rendered, based on the desired density of stars in that space and a random seed value, an appropriate subset of the large vertex buffer is rendered (OpenGL allows selectively drawing subsets of geometry buffers.) This approach produces visual results humanly indistinguishable from completely random star position populations, but with zero runtime overhead associated with shuffling star positions as desired according to spatial random seed value.

Star Rendering / Shading

Stars are rendered using a simple fragment shader applied to a camera-facing billboard quad. The fragment shader was artistically tuned to achieve the desired results, with some physical / mathematical inspiration as well. It works by simply scaling the intensity of light using inverse distance squared from the center of the billboard quad in view space. Specifically, the equation used is:

$$L_{\text{rgb}} = \min[C_{\text{rgb}} / (\Delta x^2 + \Delta y^2), 1]$$

where “min” computes component-wise minimum, and C_{rgb} is an extremely high dynamic range light value representing the intensity and color of the star (usually around ~ 100 in magnitude, where 1.0 is screen pixel color component maximum). This equation creates a visually appealing radial “glow” effect, as seen in Figure 14.

The min function used here represents the GPU’s automatic color saturation behavior; when an output color value exceeds the maximum value 1.0 for each component (red, green, and blue), it is simply clamped/saturated to that value. Note that since saturation occurs per-channel, extremely bright output values (e.g. those near the center of the star) are saturated to full white. Experiments were also done where saturation was performed first and color applied later in order to preserve color of the inner solid part of the star, but an aesthetic choice was made against this, as it did not create the same aesthetic feeling of extreme brightness.



Figure 8. Radial star glow example.

Concurrent Generation of High-Resolution Planet Datasets on the GPU

An important goal of Kosmos is that the entire universe is seamlessly interact-able, with no loading screens or unexpected freezes while new content is loaded. With a universe so large, it is of course impossible to maintain all content in memory at once. Therefore, it is important to implement a high quality concurrent content loading system. This is a difficult problem in every 3D engine, because implementing concurrent loading is a careful balance; loading too fast causes lag and stuttering, but loading too slow causes frustratingly long periods of low-resolution placeholder content being displayed.

As described in the System Architecture section, Kosmos uses a unified progressive loading cache system that facilitates this concurrency in various parts of the engine. Of particular note is how Kosmos uses this interface to implement the concurrent loading of planet datasets, which are fairly large – a single high-resolution planet dataset consumes about 500 MB of video memory, and all of this is generated concurrently in the background when the viewer approaches the planet.

First, note that the concurrent content cache enables progressive loading but does not enforce any particular rate at which content is concurrently loaded. Therefore, Kosmos implements a variable rate loading system where an in-progress load job can be sped up or slowed down at any time. Specifically, the loading speed is dynamically adjusted based on the viewer's proximity to the planet. When the user is viewing the planet from a long distance and approaching slowly, the regular load rate is used – this reduces loading lag to indistinguishable levels. As the user gets extremely close (in danger of seeing blurriness from the low resolution data set), the load speed is automatically multiplied 2x to ensure the best visual results on screen.

The actual generation of planet content itself is accomplished with a very complex GLSL fragment shader applied to a fullscreen quad, with the framebuffer configured to write the results to a RGBA texture in VRAM. The incremental loading is accomplished by adjusting the OpenGL viewport and scissors to only a fraction of the whole vertical range, therefore effectively performing exactly as much work as desired. (The partial data structure returned to the concurrent content cache represents how much of the vertical

range has been already rendered, and intelligently chooses the next vertical region when starting the next load step.) Note that when issuing the draw call for this operation, OpenGL is smart enough to allow it to run in parallel with the rest of the ordinary render process; as long as no other render operations depend on the output from such a draw call (and they don't since the texture is not used until fully complete), it is executed in parallel, and therefore does not disturb regular render process latency.

Note that there are actually two stages of generation for planet datasets. In the first stage, only the heightmap of a planet cube face is generated. Heightmaps are initially generated to an internal temporary framebuffer with a single fixed precision component (for the height values). Then, a different shader generates normal map values from this heightmap (applying Sharp Normal Map enhancement, as described in previous sections) while rendering the final results to a more permanent four-channel *RGBA* render texture target. The *RGB* components of this texture store the sharp normal map, while the *A* component stores the terrain height. This data is later used by the planet terrain rendering algorithm (SCSLOD) in a vertex shader via vertex texture fetch.

SOLVING PRECISION LIMITATIONS

Emulated 128 Bit Coordinates

Due to the massive size of the virtual universe in Kosmos, even 64 bit floating point precision is not nearly enough to represent the range of scopes necessary with sufficient precision. Ideally, one would use 128 bit fixed-point coordinates consisting of two 64 bit components (integer and fractional), but unfortunately JavaScript is a poorly designed language and does not support integer types *at all*. Still, a solution was necessary.

Kosmos emulates a high precision coordinate system using pairs of two 64 bit floating point values (the only numeric type available in JavaScript). This is configured such that a scalar coordinate $x = x_a + x_b$, where x_a contains the integer component and x_b the fractional. Adding two numbers together therefore consists of adding their integer parts and fractional parts, etc. Note however that it's important to correctly match integer parts with integer parts when constructing a new summed scalar, because due to floating point prevision behavior, mixing up the component magnitudes would result in the fractional parts being rounded off.

Using Floating Origin on the GPU

Even with “pseudo-128-bit precision” emulated from JavaScript, one still encounters a serious limitation of modern GPUs: The vast majority of GPUs only support 32-bit floating point values when performing OpenGL 3D rendering. Even though some GPUs support double precision 64 bit floats via compute frameworks (e.g. CUDA or OpenCL), this does not benefit OpenGL, and typically using such larger floats comes at a significant performance impact. Therefore, Kosmos required a way to render a universe consisting of an exponential scale of scope ranging 128-bit coordinates using only 32-bit GPU computations.

In general this is impossible, but fortunately perspective projection provides a special case where the theoretical issue can be avoided. Simply put, since objects in the distance are seen much smaller than objects near the viewer, distant objects do not need as much

precision as those very close. Theoretically, as long as the destination screen resolution (with supersampling / anti-aliasing taken into account) does not exceed 32-bit precision, there should be some way to achieve sufficient precision.

The solution in this case is to use a floating origin system on the GPU. Specifically, all render operations are performed relative to the viewer. So, whenever the CPU sends the GPU model/view/projection matrices, the model and view matrices must be pre-multiplied. Another important consideration is that GPU programs must take care in general not to inadvertently transform coordinate values to great distances, then back again, since this will cause floating point precision issues such as jittering output.

z-Buffer Precision Concerns and Mitigation: Layered Depth Rendering

Yet another limitation of modern GPU hardware is the restriction to 24-bit or 32-bit depth buffers. Depth buffers (commonly referred to as z-buffers) are necessary for the 3D objects to self-occlude, i.e. for shapes near the viewer to correctly conceal shapes that appear behind. Unfortunately, z-buffer precision is not particularly high, with most GPUs supporting only 24-bits (the remaining 8 bits usually dedicated to stencil buffer operations). This is usually not a problem, since most simulations and video games do not have a depth range so vast that this becomes an issue. However in Kosmos, a given frame may need to render distances ranging from meters to light-years, simultaneously. This scope easily exceeds the capacity for a 24-bit or even 32-bit depth buffer to retain useful precision, therefore causing serious tearing and jagged artifacts on the screen.

Kosmos implements a layered rendering approach to solve this problem. Specifically, rather than rendering the scene all at once with the same depth buffer, several render passes are made. First, stars and distant planet dots are rendered with the z-buffer disabled (since star shading is additive and therefore commutative). Then, low-resolution distant planets are sorted from farthest to nearest, and rendered with also z-buffer disabled. This technique still produces the desired occlusion effect, as long as no two planets physically intersect one another (depth sorting is an older method of object occlusion, and works very well here since it doesn't require a z-buffer). Finally, high-resolution planets are rendered with z-buffer enabled. Moreover, the near and far planes

of the view frustum are optimally set every frame based on the nearest and farthest visible point on the high-resolution planet mesh.

USER INTERFACE

The user interface in Kosmos is extremely minimalist. It consists simply of a large slider bar on the left to control speed, with three buttons: Autopilot, Reverse, and Share.

Introduction Screen

When the user first loads Kosmos, they are greeted with a simple welcome message providing all necessary instructions in a single sentence (see Figure 9).

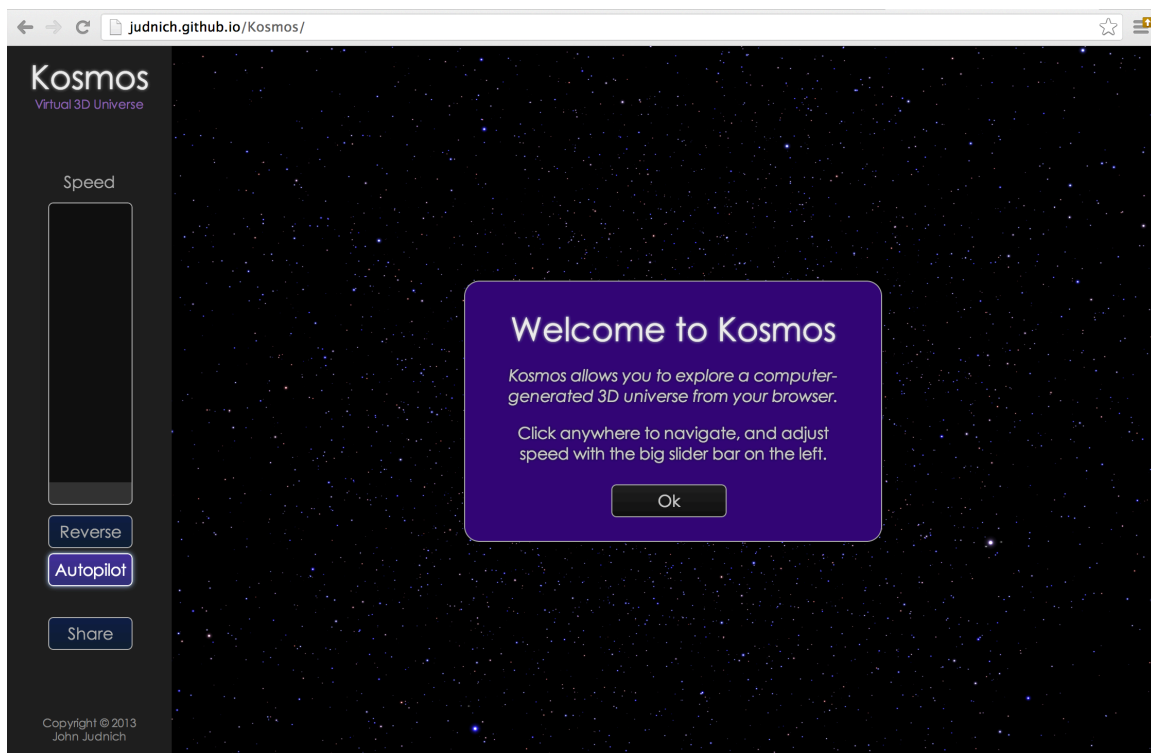


Figure 9. Kosmos introduction screen.

Autopilot Button: “Intelligently” Aided View Controls

When approaching a planet from space, the viewer’s roll orientation is not guaranteed to be “upright” from the planet’s perspective, since “upright” is of course relative to what is considered “down” and there is no such thing as “down” in deep space. However, since Kosmos’s view controls do not provide a specific method of controlling roll orientation, it’s important to support a feature where the camera is automatically oriented upright

when the viewer approaches a planet. This behavior is accomplished by converting the camera's orientation quaternion to a 3x3 rotation matrix, correcting the up vector, re-orthogonalizing, converting back to quaternion, and finally performing a smooth spherical linear interpolation on the quaternions to animate the rotation smoothly.

In addition to automatic orientation controls, another control issue encountered was difficulty in adjusting viewer speed due to the vast exponential scale of the universe. In particular, the difference in speed / scale between navigating the surface of a planet, versus traveling between planets, versus traveling between stars are all separated by many orders of magnitude. While the default speed slider control is on an exponential scale, it's still difficult to precisely set the appropriate speed. Kosmos solves this issue by providing an "Autopilot" feature (enabled by default) where the viewer's speed is automatically scaled relative to one's proximity to planetary bodies. As one approaches a planet for example with the speed bar at a constant setting, the view speed will automatically decelerate down to a gentle approach vector.

Share Button

When the user discovers an interesting object or view, it would be useful to be able to share the discovery by externally referencing the desired location / view orientation. Kosmos implements an interface where global view orientation and position coordinates may be provided as a part of the URL, allowing any location / view of the universe to be directly hyperlinked. Most importantly, a simple "Share" button is provided, which when clicked, displays a text box containing a link to what the user is currently seeing.

This link can then be copied and pasted to the user's favorite social networking site, email, chat, or any other communication medium. This copy-paste method was preferred to the traditional "Share on [insert social network here]" method because it is simpler and more universal.

Location Persistence

When the user stops using Kosmos and returns later, it would be undesirable if the view location and orientation were reset to a default every time. As a result, Kosmos periodically saves the viewer's location and orientation to persistent storage on the client side (specifically, Kosmos uses HTML5's localStorage feature). When started, Kosmos automatically checks this local storage and restores the saved location if present.

Browser Compatibility Warning

Unfortunately, Kosmos encountered a number of major compatibility issues where WebGL (the most important API Kosmos uses) behaves inconsistently on some browsers and operating systems. In addition to attempting to mitigate the issues themselves and work around browser glitches (more details are provided on these issues in the Conclusion section below), Kosmos will automatically display a pop-up warning message informing the user if his/her operating system or browser is known to be problematic with advanced WebGL. An option to click a "Try Anyway" button allows the user to ignore the warning and attempt to run Kosmos.

Data File Format

Loading lightweight configuration options and media from external files is a very important capability that applications require to separate configuration from implementation. Therefore a simple, consistent, and high performance data format is desirable.

The author presents a new markup language, called Tuple Markup Language. TML is an extremely simple all-purpose markup language: nested lists with bracket-minimizing syntax. It enables YAML-like and XML-like semantics within the same clean and consistent language, plus much more.

TML is released as open source and other developers are encouraged to use it. Also, full documentation and examples are available at: <https://github.com/judnich/TupleMarkup>. The following is a simple example of TML demonstrating markup semantics:

```
[html |
  Hello. This is an example [b|language] test.
  [ div [class testc] | And this text is enclosed in a div. ]
  [ a [href google.com] | Click this link [i|now] ]
]
```

Compare to equivalent HTML/XML:

```
<html>
  Hello. This is an example <b>language</b> test.
  <div class='testc'> And this text is enclosed in a div. </div>
  <a href='google.com'> Click this link <i>now</i> </a>
</html>
```

For the author's purposes, TML will be primarily used for semantics involving trees of key-value pairs (configuration files, etc). The author personally prefers TML because its syntax is cleaner and simpler than XML, JSON, YAML, or most other generalized data formats currently available.

SOCIETAL ISSUES

Earlier versions of Kosmos were meant to offer a subtle educational component by representing realistic interplanetary and interstellar scales. However this recent rewrite of Kosmos has since changed to target lower end computer hardware, and now uses unrealistic scales and planet sizes for artistic and technical reasons.

Kosmos is therefore a technical demo and “toy” of sorts. Since Kosmos shares the same societal purpose and issues as any other nonviolent video game or toy, this paper omits redundant commentary on this well-known topic.

Open-Source Decision

Releasing this project as open-source has several advantages not easily derived from closed source, secretive, or highly protected proprietary work:

1. The world benefits by having free access/rights to use the technology.
2. The author’s reputation benefits since open-source projects tend to spread around programmers’ social networks online much more broadly than with closed-source.
3. The author benefits because recruiters and tech companies have a real-world work sample from which to make a confident hiring/recruiting decision.

As a result of open-sourcing a project, the world benefits as a whole from the free contribution of technology. In exchange, the author gains higher visibility/popularity in the online software development community. Open-sourcing high quality projects builds reputation and connections, and by extension, one’s career.

Moreover, the author believes software patents to be anti-competitive and anti-innovative in most cases. While some of the algorithms could have been patented (for example, “Sharp Normal Maps”), doing so would only discourage the industry to use it and improve upon it. Although promoting and licensing such a patented technology could lead to financial gain, doing so would still require managerial time and effort perhaps

better spent elsewhere. (There are of course exceptions; for example if the author starts a business critically dependent on a particular secret algorithm, it would be imperative to legally protect the intellectual property.)

By releasing a technology to everyone, its use becomes more pervasive through the industry (since there is less financial resistance). As a result, one may still benefit financially if only indirectly, as one's reputation grows and further employment and entrepreneurial opportunities open in the longer-term future.

CONCLUSION

Despite major changes to the project goals requiring a complete rewrite within a few months of the final design conference deadline, all the primary goals were achieved: Kosmos allows you to explore a computer-generated 3D universe containing trillions of stars, planets, and moons – right from your web browser. Moreover, this latest incarnation of Kosmos does not require an extremely powerful gaming PC to run; most *modern* laptops should be able to run Kosmos smoothly.

Lessons Learned

1. WebGL implementations are flakey and not ready for high-resolution 3D games yet.

During the development of the latest version of Kosmos, a number of compatibility issues were encountered with the current browsers' WebGL implementations. In particular, Kosmos seems to run best on Mozilla Firefox on Mac or Linux computers. Chrome on Mac encounters strange performance issues with no explanation or warning, likely due to silent software rendering fallbacks, while Firefox instead uses the GPU as expected.

What's worse, on Windows Kosmos initially did not work at all (even in Firefox). Upon debugging the issue, Windows seems to use Google's ANGLE, a so-called "compatibility layer" for WebGL used in both Firefox and Chrome. ANGLE converts WebGL calls (which are really just OpenGL calls exposed to JavaScript) into DirectX. This conversion means ANGLE recompiles OpenGL's GLSL shaders into DirectX's HLSL language. In theory, this translation process would be fine if it worked – but it's very buggy.

The very advanced GLSL shaders in Kosmos encounter what seem like unanticipated / untested edge cases in Google's ANGLE compatibility layer, because Chrome crashes on Windows and Firefox froze and failed to load planet data at all (due to failure to correctly execute the corresponding GLSL shaders). Fortunately, with much trial and error, removing some features to simplify GLSL shaders a bit brought Kosmos to a point where it works on Windows, though not without compromise (in particular, Windows cannot support an infinite variety of planets that works just fine on Mac and Linux).

Even with compatibility issues aside, Google's ANGLE is extremely slow in processing GLSL shaders. As a result, Kosmos loads very slowly on even powerful windows computers because the 3D engine uses very large and complex GLSL shaders to offload as much work to the GPU as possible. In contrast, on Mac and Linux where ANGLE is not enabled, Kosmos loads in under 2 seconds (even on a laptop an order of magnitude slower than the Windows computer tested).

2. Procedural content generation, while a nice idea in concept, ultimately doesn't "save" you all that much work. In theory it provides "infinite variation" of planets, stars, etc., but it does not provide infinite novelty.

What the human mind finds interesting artistically and visually is not variation, but "novelty". While admittedly a more vague word, novelty represents content that is truly new, rather than just parameterized variations of the same thing seen before.

Although procedural generation engines (like in Kosmos) can provide infinitely varying universes with trillions of stars without the need for each to be individually designed, it becomes boring after a while because human minds adapt and figure out the underlying patterns very rapidly.

Therefore, some amount of handcrafting and artistically created content is needed to make content sufficiently interesting for a game, for example.

However, one possible exception would be a more intricate simulation-oriented generation system (i.e. rather than using simple mathematical functions to generate planet-resembling things, actually simulate gasses in space, gravity in space, star formation, planet formation, erosion, elements, etc. etc.) Such a simulation could produce a system so complex that variation does appear truly "novel" in some sense, simply from the sheer scale and detail of the simulation. However such complexity would require far too much computational power to be feasible in real-time, even on powerful gaming computers, let alone mobile/casual devices. It might be feasible though as an offline content creation tool to ease the work of artists, however. Or, it could be feasible if

offloaded to a supercomputing "cloud", with the data streamed to users of the game world on-demand. Even then though, creating the rules for such a simulation would be no small feat in of itself.

So aside from major advances in simulation complexity (which may happen in a few decades), it appears most 3D worlds will need to have some aspect of human-guided design to be effectively interesting for games. This observation by no means rules out all procedurally generated content, it just means one will likely spend about as much time crafting procedural rules/equations as an artist would making it by hand anyway.

Future Plans

Kosmos (this web based version version) was mostly an experimental, self-educational project for the author. In retrospect, WebGL simply has too many compatibility issues/hassles for it to be a valuable target for spare time projects. Future versions or projects will most likely be done with native code instead -- certainly until WebGL stops being flakey, and browsers figure out how to get closer to native performance.

For example, the author would like to make a future improved version of Kosmos as a mobile game app, targeting tablets in particular. Additionally, there are a lot of features that would be included that weren't possible in this version due to time constraints:

- Planet atmospheres with correct simulated atmospheric scattering effects
- Much more ground-level detail (i.e. trees, grass, etc.)
- Animated planet orbits and rotations
- More variety of planet types (right now there's just a few base types)
- Gameplay dynamics with space and ground combat

Of course, the completion of such future goals is subject to time constraints, since the author will be employed working full-time on other projects following graduation.

REFERENCES

- [1] John Judnich. *Kosmos (Full Source Code)*. 2013. <github.com/judnich/Kosmos>
- [2] Vladimir Romanyuk. *Space Engine*. 2013. <<http://en.spaceengine.org>>
- [3] John Judnich and Nam Ling, “Fast Multiresolution Terrain Rendering with Symmetric Cluster Sets,” *Proceedings of the 4th ACM SIGGRAPH Conference and Exhibition on Computer Graphics and Interactive Techniques in Asia (SIGGRAPH Asia)*, Hong Kong, China, December 12 – 15, 2011.
- [4] John Judnich and Nam Ling, “Symmetric Cluster Set Level of Detail for Real-Time Terrain Rendering,” *Proceedings of the 2012 IEEE International Conference on Multimedia and Expo (ICME)*, Melbourne, Australia, pp. 320 – 324, July 9 – 13, 2012.
- [5] R. Pajarola and E. Gobbetti, “Survey of semi-regular multiresolution models for interactive terrain rendering,” *Vis. Comput.* vol. 23, pp. 583–605, July 2007.
- [6] F. Strugar, “Continuous distance-dependent level of detail for rendering heightmaps,” *Journal of Graphics, GPU, & Game Tools*, vol. 14, pp. 54–74, 2009.
- [7] John Judnich and Nam Ling, “Fast Per-Pixel Displacement Mapping with Screen-Space Depth Interval Grids,” under review for publication. <<http://judnich.github.io>>
- [8] L. Szirmay-Kalos and T. Umenhoffer, “Displacement mapping on the GPU: State of the art,” in *Computer Graphics Forum*, vol. 27, no. 6. Wiley Online Library, 2008, pp. 1567–1592.

APPENDIX

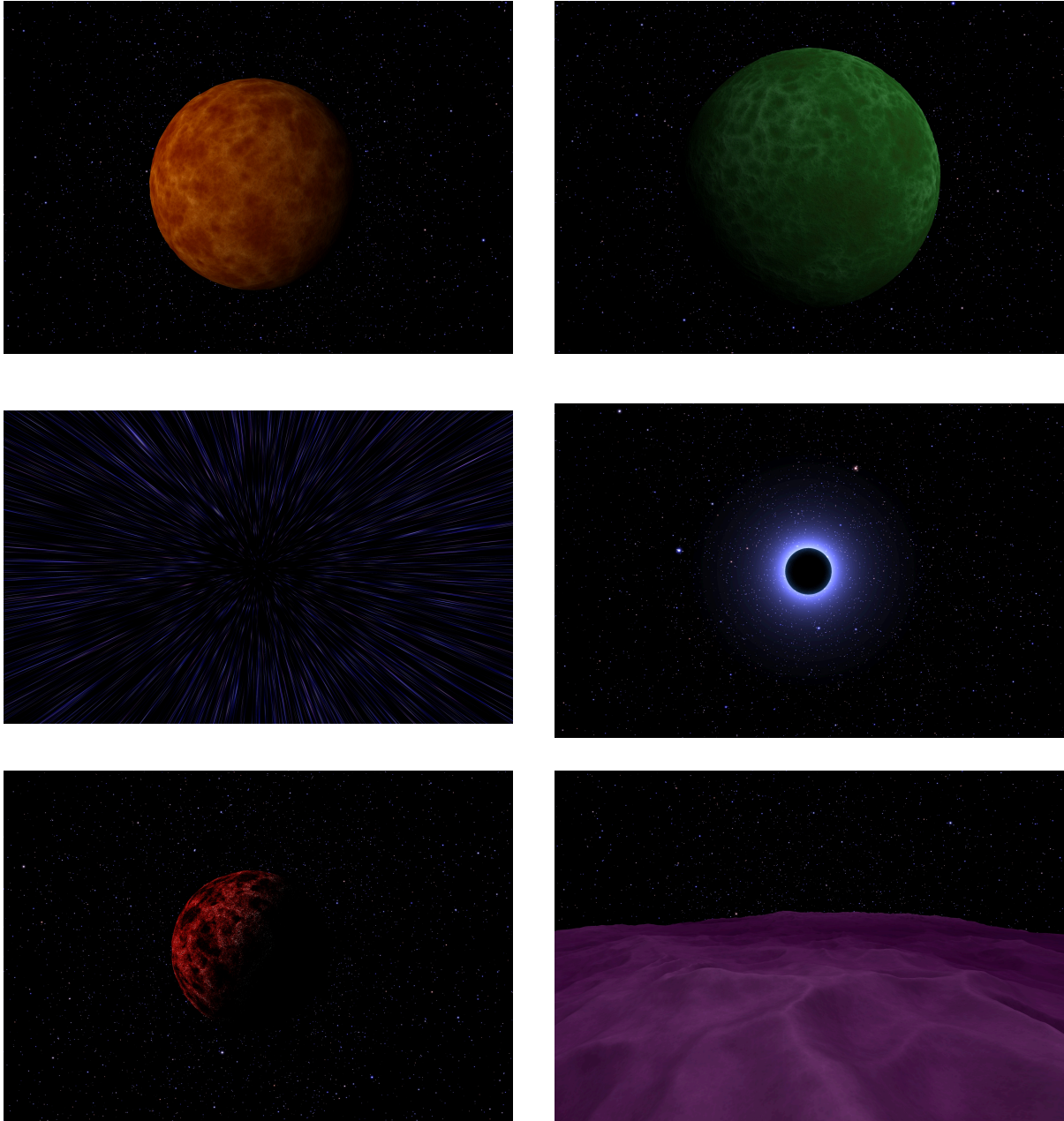


Figure 10. Additional screenshots.