SN-Engine, Geometric Modelling Environment for Virtual World Design and Perception

Timur A. Zhukov [0000-0002-0600-2037]

Plekhanov Russian University of Economics, 36 Stremyanny lane, Moscow, 115998, Russia
zhukov.ta@rea.ru

Abstract. We present a new scale-free geometric modelling environment designed by the author of the paper. It allows one to consistently treat geometric objects of arbitrary size and offers extensive analytic and computational support for visualization of both real and artificial sceneries.

Keywords: Geometric modelling, Virtual reality, Game engine, 3d visualization, Procedural generation.

1 Introduction

Geometric modelling of real-world and imaginary objects is an important task that is ubiquitous in modern computer science. Today, geometric modelling environments (GME) are widely used in cartography [11], architecture [26], geology [12], hydrology [24], and astronomy [27]. Apart from being used for modelling, importing and storing real-world data, GMEs are crucially important in computer games industry [17].

Creating a detailed model of a real-world or imaginary environment is a task of great computational complexity. Modelling various physical phenomena like gravity, atmospheric light scattering, terrain weathering, and erosion processes requires full use of modern computer algebra algorithms as well as their efficient implementation. Finding exact and numeric solutions to differential, difference, and algebraic equations by means of computer algebra methods (see [14] and the references therein) is at the core of the crucial algorithms of modern geometric modelling environments.

There exist a wide variety of geometric modelling environments, both universal and highly specialized. They include three-dimensional planetariums like SpaceEngine or Celestia, virtual Earth engine Outerra, procedural scenery generator Terragen, 3D computer graphics editors Autodesk 3ds Max, Blender, Houdini, and 3D game engines like Unreal Engine 4, CryEngine 3, and Unity.

The present paper describes a new geometric modelling environment, SN-Engine, designed and implemented by the author of the paper. The main advantage of this GME is its capacity to consistently and in a computationally efficient way treat geometric objects of arbitrary size, from extragalactic and down to the microscopic scale.

Proceedings of the 10th International Scientific and Practical Conference named after A. I. Kitov "Information Technologies and Mathematical Methods in Economics and Management (IT&MM-2020)", October 15-16, 2020, Moscow, Russia

This geometric modelling environment is a freeware implemented in C# programming language. Scripts, sample video files and high resolution images created with SN-Engine are publicly available at snengine.tumblr.com.

2 State-of-the-art in geometric modelling: capabilities and limitations

The geometric modelling environment presented in the paper shares many properties and functions with other GMEs. These include procedural generation, instruments and algorithms of 3D rendering, internal elements of system architecture, file and in-memory data formats, etc. In the Tables 1, 2, 3 we compare SN-Engine with other existing systems.

Table 1. Space modelling systems.

Title	Type	Procedurally generated objects	Planet surfaces
SpaceEngine	astronomy program, game engine	galaxies, star systems, planets	detailed surface, no ground objects
Elite Dangerous	game	galaxy, star systems, planets	detailed surface, no atmospheric planets
Star Citizen	game, publicly available alpha version	planet details, object layouts	detailed surface, ground objects, no true-scale planets
No Man's Sky	game	planets, objects, species	detailed voxel terrain, ground objects, no true-scale planets
Celestia	astronomy program	none	textured model
Outerra	3D planetary engine	planet surface details, objects	detailed surface, biomes, Earth data, ground objects
SN-Engine	world platform/game engine	galaxies, star systems, planets, objects	detailed surface, biomes, Earth data, ground objects

Despite the extensive capabilities of modern geometric modelling systems all of them have limitations in terms of world structure, engine modification, userworld interaction, and licensing. The presented modelling system SN-Engine is a freeware which, unlike any other software listed above, is endowed with tools for procedural generation of objects of arbitrary scale.

Table 2. 3D computer graphics editors.

Title	License	Use	Type of procedural generation

Blender	free	3D computer graphics, mixed	scripted
Autodesk 3ds Max	commercial	3D computer graphics, mixed	scripted
Autodesk AutoCAD	commercial	computer-aided design and drafting	partial, scripted
Houdini	commercial	3D animation, mixed	integrated, scripted
CityEngine	commercial	urban environments generation	integrated
Terragen	commercial, free	procedural landscape generation and visualization	integrated
SN-Engine	no public release yet	procedural world creation	integrated, scripted

Table 3. 3D game engines.

Title	Game creation methods	Game creation license	Source availability	Type of procedural generation
Unreal Engine 4	editor, visual programming, C++ programming	free / royalty for commercial use	open	plugins, scripted
Unity	editor, C# programming, prefab construction	free / subscription model	closed / open on enterprise license	plugins, scripted
CryEngine 3	editor, C++ programming	free / royalty for commercial use	open	plugins, scripted
SN-Engine	editor, Lua programming, C# programming	free	open scripts, closed source code	intergated, scripted

The presented geometric modelling system unifies procedural approach to content generation and game engine technologies to create a fully flexible multiclient, arbitrary scale world creation and experience platform. The system can be used for solving various tasks like demonstration, design, recreation, and education. Its functions include generation of new content, transformation of existing content, and visualization. By organizational structure the presented geometric modelling system can be used in autonomous, local-networked or global-networked mode.

The engine provides flexibility in terms of world and entity modification, creation and generation. The list of main features is as follows:

– Support of arbitrary scale worlds, from super galactic to microscopic level;
– Client-server world and logic synchronization;
– Scriptable game logic;

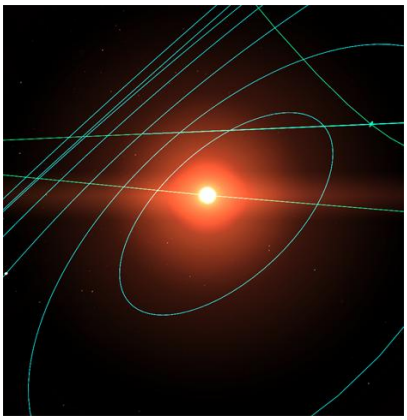
– Procedural generation [17, 18, 21, 25] of any world content ranging from textures and models, to full world generation;
– External data import: e.g. height maps [2, 23], OpenStreetMap [5] data, textures, models, etc.;
– Integrated computational physics module;
– World state system for saving world data including any runtime changes;
– Script system controlling every world related function of the engine;
– Fully extendable and modifiable library of objects including galaxies, planets, lights, static and dynamic objects, items and, others;
– Support of player controlled or computer controlled characters

The base engine realization contains a procedurally generated universe with standard node hierarchy (Fig. 1) ranging from galaxies to planet surfaces and their objects. Hierarchy is displayed in columns from left to right and from top to bottom: galaxies (a), spiral galaxy stars and dust (b), a star system, planets and their orbits (enabled for clarity) (c), an earth-like planet (d), planet surface seen from a low orbit (e), planet surface from ground level (f). In addition, other nodal hierarchies can be created with custom nodes by means of procedural generation algorithms.

A planet has a touchable surface endowed with physical properties which satisfy the laws of the physics of solids. Players can walk on a planet's surface and interact with other objects.

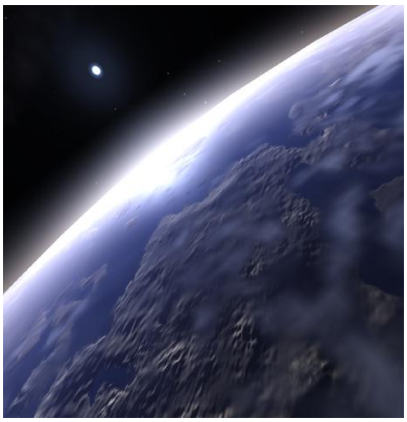
(a) A spiral supergalactic system

(b) A single galactic

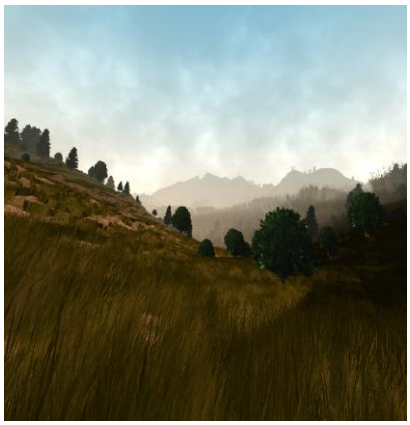


(c) A planetary system

(d) An earth-like planet from the orbit



(e) A planet seen from the stratosphere



(f) The terrain of a planet

Fig. 1. The default world hierarchy in SN-Engine environment

3 Implementation of the geometric modelling engine

The engine is implemented in C# on .NET Framework and uses Lua [4] as the main scripting language. The engine consists of core systems and modules.
The core systems are as follows:

– Task system managing task creation, processing, linking and balance;

– IO system or virtual file system, providing synchronous and asynchronous asset loading, reloading, and dependency management;

– Object system or global object management system, base for Assets, Nodes, Components, Events, and others;

– Event system that manages subscription and invocation of functions;

– Physics system handles physical interactions of nodes with instances of physics entities;

– Sound system supports playback of sound effects and ambience in 3D space;

– Horizon system providing frame of reference update and loading of nodes;

– Input system or user input engine interface;

– Net system that enables client-server event relay;

– Rendering system which does visualization of nodes with attached cDrawable component;

– Scripting system providing safe and real-time game logic programming in Lua.
The list of modules comprises the following:

– Profile which stores user data, settings, saves, etc.;

– Add-on or user data package manager;

– GUI that renders user interface objects and their rendering;

– ModelBuilder providing classes and functions for dynamic model construction;

– ProcGen or operation based procedural generation system;

– sGenerator or script based procedural generation system;

– Flow, the node based visual programming language;

– Forms, the dynamic index database of user defined data assets;

– VR, the virtual reality Oculus HMD interface;

– Web module that provides Awesomium browser interface, rendering to texture and control functions.

Data modules include primary asset modules which contain methods for asset manipulation in JSON format [3], Material, StaticModel, Particles, Sprite font, SurfaceDataStorage, Package, Texture, Sound, and secondary format modules that are used in data import stage: SMD, FBX, BSP, OSM, MCV

3.1 Hierarchical Nodal System of the Geometric Modelling Engine

The core component of the engine world structure is a node. Every node follows the set of rules:

– A node can have space bounds with box, sphere or compound shape;

– A node can contain any finite number of other nodes and one parent node, forming a tree graph structure;

– A node has absolute size variable measured in meters per node unit, so in case of node with spherical bounds its radius is equal to the absolute size of the node;

– A node has position, rotation and scale variables which define its location and scale in its parent node;

– A node has seed variable used for procedural generation;

– Nodes can have components, custom variables and event listeners;

– A node without parent node is the world node.

A possible node type hierarchy of a client camera flying two meters above the planet surface is as follows: 0) world sol; 1) spacecluster; 2) galaxy; 3) starsystem; 4) star; 5) planet; 6) planet surface; 7) planet surface node; 8) camera. All types in this hierarchy, with the exception for the camera type, are defined in script files.

The location variables and node hierarchy allows one to build relative transformation matrices and their inverse matrices for each parent node in its hierarchy. This in turn allows one to obtain relative transformations for any node within the same world.

The precision of 32 bit floating point numbers which are widely used in computer graphics is limited while the corresponding precision distribution is not linear. Any number with 7 or more significant decimal digits is subject to data loss during mathematical operations. There are several workaround methods to overcome this limitation and each method needs to calculate object positions in relation to camera at runtime. These methods include the following:

– Storing object positions as 64bit floats, which raises overall precision to 15 decimal digits but still has the same precision distribution limitations;

– Storing object positions as 32bit or 64bit integers, since integers have linear precision distribution;

– Storing object positions in relation to specialized group object. This method is used in most geometric modelling systems;

– Storing relative local object positions with respect to their parent object. This method requires hierarchical organization of objects.

We use the last method to solve this problem. This is done in three steps. The first two steps are performed when creating or updating a node while the third is performed at the rendering stage.

The first step is to compute the node world matrix, which consists of the node scale, rotation and position:

$$W = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1-2(R_y^2+R_x^2) & 2(R_xR_y+R_zR_w) & 2(R_zR_x-R_yR_w) & 0 \\ 2(R_xR_y-R_zR_w) & 1-2(R_z^2+R_x^2) & 2(R_yR_z+R_xR_w) & 0 \\ 2(R_zR_x+R_yR_w) & 2(R_yR_z-R_xR_w) & 1-2(R_y^2+R_x^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times$$

$$\times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ P_x & P_y & P_z & 1 \end{pmatrix},$$

that is, W = scale (S) · rotation (R) · translation(P). Here W is the node world matrix, $S = (S_x, S_y, S_z)$ is the node scaling vector, $R = (R_x, R_y, R_z, R_w)$ is the node rotation quaternion, and $P = (P_x, P_y, P_z)$is the node position vector. The second step is to

compute hierarchical matrices using world matrices obtained at the previous step and parent nodes:

$$H_i = \begin{cases} Id_4, & \text{if } i = 0, \\ H_{i-1}\dfrac{S_{i-1}}{S_i}W_i, & \text{if } i > 0. \end{cases}$$

Here

– Id_4 is the identity matrix of size 4;

– i is the node level in parent hierarchy, where 0 = node, 1 = parent, 2 = parent of parent, ... , n = world node;

– H_i is the i-th level node world transformation matrix;

– S_i is the i-th level parent node absolute node size in meters;

– W_i is the i-th level parent node world matrix.

Finally, the third step is the local world matrix computation (Fig. 2, where (a) is the current node, (b) is the target node, (c) is local world matrix and (d) is the closest common ancestor for the current and the target nodes). This matrix is computed as follows:

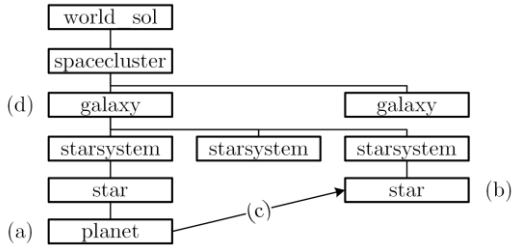


Fig. 2. Node local world matrix computation

$$LW_N = H_{(C_{level} - T_{level})} H^{-1}_{(N_{level} - T_{level})}$$

Here LW is the local world matrix of node N at node C, T (the top node) is the nearest node in hierarchy to both current and target nodes, C is the current node, N is the target node, level is the node hierarchy level starting from the world node.

3.2 System of the Components

The engine nodes use partial implementation of Entity component system [10] pattern with the difference that components can have their own methods. The nodes are entities in this pattern. Components define how a node interacts with other nodes, how it is rendered and how it functions.

Core engine components (*sComponent*) include the following: Drawable object components (*cDrawable*): *cModel* for model instances, *cParticleSystem* for particle system instances, *cSkybox* for skybox instances, *cSurface* for planet surface (terrain and water) instances, *cSpriteText* for text sprites in 3D space, *cAtmosphere* for atmospheric fog, *cVolume* for volumetric texture renderer instances.

Update phase components (*cUpdater*): *cOrbit* which sets node position using orbital parameters and current time, *cConstantRotation* that sets node rotation from current time.

Physics components (include interfaces for physics engine [1]): *cStaticCollision* or collision mesh with infinite mass, *cPhysicsSpace* for physical space which contain physics objects, *cPhysicsObject* for physics objects with finite mass and volume, *cPhysicsMesh* or concave triangulated physics mesh, *cPhysicsCompound* or compound physics object, *cPhysicsActorHull* or physical controller for actors.

Content generation components: *cRenderer* components that perform render to texture operations, *cCamera* which renders from camera node, *cCubemap* that renders 6-sided cubemap texture, *cHeightmap* draws to planetary height map texture, cInterface renders hierarchy of panel objects to GUI texture, *cShadow* draws scene to cascading shadow map [15] texture, *cProcedural* or procedural node generation component.

Other component types: *cLightSource* point light which is used for illumination, cNavigation which generates and contains navigational map for AI actors, *cPartition* is hierarchical space partitioning (part of HorisonSystem), *cPartition2D* or quad tree partitioning (4 subspaces on 2 axes), *cPartition3D* or octree partitioning (8 subspaces on 3 axes), *cSurfaceMod* is surface data (height, temperature, etc.) modifier (*cSurface*), *cWebInterface* is web interface data container.

Orbital component
uses Keplerian orbits to setup its node position. When an orbit is created, we calculate the average orbital speed $V_{orbital}$ in terms of of masses and semi-major axis length using the formula

$$V_{orbital} = \sqrt{G\frac{M_{node}M_{parent}}{a^3}}.$$

Here G is the gravitational constant, a is the semi-major axis, M_{node} is the orbiting node mass, M_{parent} is the parent node mass. Subsequently in the update event we use Algorithm 1 to calculate the vector position along the shifted ellipse and then rotates it around 0 with the argument of periapsis along the Y axis, inclination along the X axis and ascending longitude along the Y axis.

Algorithm 1 orbital position update step

function GetOrbitalPosition$(a, b, e, p, i, l, V_{orbital}, t)$

$\hat{t} = \mathrm{mod}(tV_{orbital}, 2\pi) - \pi$

$\bar{P}_{local} = \langle -a\cos\big(\mathrm{GetCorrectedT}(e, \hat{t})\big), 0, \mathrm{sign}(\hat{t})\sqrt{(1-\frac{x^2}{a^2})b^2}\rangle$

$$\bar{P}_{parent} = \bar{P}_{local}\begin{pmatrix}\cos p & 0 & \sin p\\ 0 & 1 & 0\\ -\sin p & 0 & \cos p\end{pmatrix}\begin{pmatrix}1 & 0 & 0\\ 0 & \cos i & -\sin i\\ 0 & \sin i & \cos p\end{pmatrix}\begin{pmatrix}\cos l & 0 & \sin l\\ 0 & 1 & 0\\ -\sin l & 0 & \cos l\end{pmatrix}$$

return $\bar{P}_{parent}$

end function

function GetCorrectedT($e, \hat{t}$)
  $D = 1$
  $E = \hat{t}$
  while $D > \epsilon$ **do**
    $\tilde{E} = \mathrm{e}\sin E - \hat{t}$
    $D = \left|\tilde{E} - E\right|$
    $E = \tilde{E}$
  end while
  return E
end function

Here a is the semi-major axis, b is the semi-minor axis, e is the eccentricity, p is the argument of periapsis, i is the inclination, l is the ascending longitude, t is time, $\hat{t}$ is the orbital position. The value ϵ = 0.001 in parent node units is used as a precision limiter.

3.3 The Horizon System

The horizon system provides dynamic recalculation of coordinate systems, space to space node transfer and the space partition update. The system checks all nodes with enabled space transfer flag against the bounds of their parent node and every other node in it. When a node is outside of its parent bounds or inside of any other node bounds, the system recalculates its position, rotation, velocity, and angular velocity and then changes its parent node.

The partition component generates dynamic tree structure from partition nodes and calls events to its host node. This allows one to create procedural generators for three-dimensional (Octree) objects like galaxy generation, galactic cluster generation and two-dimensional (Quad tree) objects like planet surfaces. It can also be used for any other type of generation with scripting.

3.4 Procedural Generation

Procedural generation is the crucial part of the engine. It allows real-time content generation from pre-defined patterns and scripts. There are different methods of procedural content generation that are implemented in the engine including:

– Node generation with scripted procedural generators in Lua or C# , e.g. cluster and galaxy generation (C#), star system generation (Lua);

– Model and node generation with operation based generation system, e.g. models of objects, buildings; generated building interior world nodes and others;

– Texture generation from Lua scripts or JSON files;

– Texture generation from cCamera and cCubemap node components;

– Surface data generation [16, 20, 21, 22] from cHeightmap node component, noise functions [9] and JSON parameters.

Operation Based Procedural Generation System.
This system allows one to define dynamic model structures and to build complex 3D models with several levels. Models are defined in JSON-like files or Lua structures by lists of consecutive operations. These operations use named groups of 3D primitives along with parameters as input data. Fig. 3 illustrates the use of procedural generation in building interiors.

The primitive types comprise the following: point as a basic structure which contains position in the local space, path as a set of points with a loop flag, and surface or polygon which contains edge (path) and other surface properties such as material and uv-matrix.

(a) (b)

(c) (d)

Fig. 3. Procedurally generated interiors

The procedural generation system includes a total of 56 different operation types. The operation types are divided into groups as follows: ”Create” operations which create sets of primitives from data, ”Extend” operations which create new primitives from existing, ”Modify” operations that alter existing primitives, ”Select” operations

which filter existing primitives or return their data, and ”Utility” operations which provide branching in generation algorithm.

An example of the use of operation is given by { type : ”inset”, from:”bt base”, out : [”bt base”, ”bt sides”], extrude : 0.4, amount: 0.5 }. It applies ”inset” operation (insets a polygon edge by the ”amount” value and shifts it by the ”extrude” value) for each primitive in the group ”bt base”. It outputs the central polygon to the ”bt base” group and the edge polygons to the ”bt sides” group.

3.5 The Rendering System

Rendering system uses SharpDX [7] library which is an open-source managed .NET wrapper of the DirectX API. The engine works on DX10 API and DX11 API.

The system uses a combination of forward and deferred rendering techniques [8, 13] to draw objects and effects. The rendering process is the most difficult one in terms of computational complexity and consists of several different stages. The pre-processing stage is when all texture generation is performed. The rendering system processes the incoming draw requests from cRenderer components and fires their respective events. Then in the main stage, the system performs successive render calls to cDrawable components, which are divided by the render groups and layers. Finally, at the post-processing stage the system combines the results of the previous draw calls into a single texture and applies screen effects including screen space local reflections [6] and screen space ambient occlusion [19].

Atmosphere rendering shader.
The atmospheric shader is a program for GPU which manages the rendering of a ”fog” layer. This shader is a work in progress. One of the key algorithms implemented in this program is as follows

Algorithm 2 Atmosphere glow and fog shader

function GetFogColour$(\mathrm{N}, \mathrm{W}_{\mathrm{world}}, \mathrm{N}_{\mathrm{planet}}, \mathrm{W}_{\mathrm{planet}}, \mathrm{W}_{\mathrm{hrz}}, \dots)$

// 1. The horizon line and gradient density calculation

$t_d = \mathrm{dot}(N, N_{planet}) + \frac{\mathrm{W_{hrz}}}{\mathrm{W_{planet}}}$

$\mathrm{D_{top}} = \mathrm{saturate}(\mathrm{t_d})$

$\mathrm{D_{hrz}} = 1 - |\mathrm{t_d}|$

$\mathrm{D_{bot}} = \mathrm{saturate}(-\mathrm{t_d})$

// 2. Atmosphere density from view distance calculation

$\mathrm{D_{world}} = 1 - \mathrm{saturate}(0.01 \mathrm{M W_{hrz} W_{world}})$

$\mathrm{D_d} = \max\left(1, \frac{\mathrm{H}}{\mathrm{W_{atmosphere}}}\right)$

// 3. Light colour and sun glare calculation

for i=0 i<n i++ **do**

$\mathrm{C_{light}} \mathrel{+}= \max(0, 1 - \mathrm{saturate}(5\ \mathrm{dot}(\mathrm{N}, \mathrm{N_i}))\mathrm{C_{atmosphere}} \mathrm{C_i})$

$\mathrm{D_i} = \max(0, \mathrm{dot}(\mathrm{N}, \mathrm{N_i}))$

$S+=(0.5D_i+0.4D_i^{10}+0.3D_i^{100}+0.2D_i^{1000})C_i$

end for

// 4. Atmosphere parts colour calculation

$C_{top}=\frac{C_{atmosphere}}{D_d}$

$C_{hrz}=\frac{C_{haze}lerp\left(0.25(C_{sun}+C_{light}),C_{light},saturate(N_l+0.1)\right)}{D_d}+\frac{saturate(N_h)}{D_d}$

// 5. Atmosphere parts density calculation

$D_a=\frac{D_{bottom}+D_{top}}{max(1,D_d)}+\frac{D_{hrz}}{D_d}$

// 6. Final atmosphere colour calculation

return $0.5D_{world}D_a((D_{hrz}+D_{bot})C_{hrz}+D_{top}(t_{back}+C_{top})+S)$

end function

Here $C_{atmosphere}$ is the atmosphere colour, C_i is the i-th star colour, C_{sun} is the star colour, C_l is the star direction, *M* is the node scale, *N* is the surface normal, N_i is the i-th star direction, N_{planet} is the direction to the planet centre, *H* is the camera distance to the planet surface towards the centre of the planet, W_{hrz} is the distance to the horizon, W_{planet} is the distance to the planet centre, W_{world} is the depth of the current pixel, $W_{atmosphere}$ is the atmosphere width, *n* is the star count, t_{back} is the pixel colour, lerp(*a*, *b*, *d*) is the linear interpolation of two vectors *a* and *b* based on the weight *d*.

The resulting image is blended with the current view. The numerical coefficients in the above formulas have been selected on the basis of visual perception.

The Scripting System

Script system uses Lua [4] as the scripting language. Scripts in Lua describe most of game logic. Types of scripts include entity definitions, player controller definitions (free camera controller, actor controller, etc.), GUI widget definitions, auto run scripts and Lua modules (definitions for user class types, structure types and libraries), procedural generator definitions and others. At the time of writing, there were 251 Lua script files in the base engine content directory with total size of 904 KB.

The Networking System

The engine network system is based on client-server model and uses packets to transfer data. The packets are being sent and received asynchronously and incoming packets placed into a queue which is then processed from the main thread in both server and clients.

4 Limitations of the presented geometric modelling environment

There exist certain limitations for each of the main systems of SN-Engine. They are summarized in the Table 4.

Hardware requirements: Processor: Dual Core CPU @ 3.0 GHz, Memory: 4GB RAM, Hard Drive: 2 GB, Graphics: Nvidia GeForce GTX 1070, 2048MB.

Software requirements: OS: Windows 7, 8, 10, .NET Framework 4.0.

Table 4. Limitations of SN-Engine modelling environment.

System	Limitations	Approximate value	Limited by
Hierarchical nodal system	maximum node nesting level	64	memory
	maximum loaded node count	tens of thousands	memory
	maximum absolute node size	$\cong 1\times10^{308}$	double type precision
	maximum simultaneous nodes in node	thousands	memory, CPU, HDD/SSD read speed
Operation based procedural generation system	polygon count	tens of millions	CPU, int32 maximum value
	complex topology	N/A	procedural operations functionality
Rendering system	maximum visible objects	thousands/ millions instances	GPU memory, GPU processing power
	maximum drawable objects	tens of thousands	CPU, memory
Task system	maximum concurrent threads	1 main thread, up to 4 background per CPU core	CPU
	maximum tasks in thread queue	thousands	CPU, memory
	maximum concurrent updatable objects	$\cong 1000$	CPU
Physics system	maximum active objects in simulation island	fast physics: $\cong$ 100 slow-motion physics: thousands	CPU
	maximum static collision triangles per physics space	millions	memory, CPU
Networking system	maximum client connections	tens	CPU, network bandwidth, memory
	maximum active networked nodes	hundreds	network bandwidth
IO System	maximum add-on count	hundreds	memory, HDD/SSD read speed
	maximum file count	tens of thousands	memory
	maximum loaded assets	varies from asset filesize $\cong 10000 - 100$	memory
Lua scripting system	processing speed(from C#)	Divided by 10-1000	CPU
	single threaded.	N/A	CPU

5 Discussion

The new geometric modelling system SN-Engine combines procedural generation algorithms, arbitrary scale nodal system, and extensive Lua scripting system to construct and visualize complex sceneries.

High resolution screenshots and video are available on the system website at http://snengine.tumblr.com/

Acknowledgements

This research was performed in the framework of the state task in the field of scientific activity of the Ministry of Science and Higher Education of the Russian Federation, project ”Development of the methodology and a software platform for the construction of digital twins, intellectual analysis and forecast of complex economic systems”, grant no. FSSW-2020-0008.

References

1. Bepuphysics: free, open source, 3d physics library. https://www.bepuphysics.com
2. Earth digital elevation map: earth heightmap data source. http://viewfinderpanoramas. org
3. Ecma-404 json data interchange format, ecma int’l. http://www.ecma-international. org/publications/files/ECMA-ST/ECMA-404.pdf
4. Lua: lightweight, multi-paradigm programming language. https://www.lua.org
5. Openstreetmap: vector geographic data source. https://www.openstreetmap.org
6. Screeen-space reflections. http://www.cse.chalmers.se/edu/course/TDA361/Advanced%20Computer%20Graphics/Screen-space%20reflections.pdf
7. Sharpdx: An open-source managed .net wrapper of the directx api. http://sharpdx.org/
8. Abbey, D.: Real-Time Rendering, 4th edn. New York: A K Peters/CRC Press (2018)
9. Bennett, M.: Frequency spectra filtering for perlin noise. The Computer Games Journal pp. 1–12 (2018)
10. Bilas, S.: A data-driven game object system. https://www.bepuphysics.com
11. Borrmann, A., Kolbe, T., Donaubauer, A., Steuer, H., Jubierre, J., Flurl, M.: Multi-scale geometric-semantic modelling of shield tunnels for gis and bim applications. ComputerAided Civil and Infrastructure Engineering 30(4), 263–281 (2014)
12. Caumon, G., Collon, P., de Veslud, L.C., C. Viseur, S., Sausse, J.: Surface-based 3d modeling of geological structures. Mathematical geosciences 41(8), 927–945 (2009). DOI 10.1007/s11004-009-9244-2
13. Debevec, P.: Rendering synthetic objects into real scenes: Bridging traditional and image based graphics with global illumination and high dynamic range photog-

raphy. In: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '98, p. 189–198. Association for Computing Machinery, New York, NY, USA (1998). DOI 10.1145/280814.280864. URL https://doi.org/10.1145/280814.280864

14. Dickenstein, A., Sadykov, T.: Bases in the solution space of the mellin system. Sbornik Mathematics 198(9), 1277–1298 (2007)

15. Dimitrov, R.: Cascaded shadow maps. http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf

16. Gagnon, J.S., Lovejoy, S., Schertzer, D.: Multifractal earth topography. Nonlinear Processes in Geophysics 13(5), 541–570 (2006). DOI 10.5194/npg-13-541-2006. URL https://npg.copernicus.org/articles/13/541/2006/

17. Kruse, J., Sosa, R., Connor, A.: Procedural urban environments for fps games. In: Proceedings of the Australasian computer science week multiconference. Australia: ACM, Canberra (2016)

18. Mark, B., Berechet, T.: Procedural 3d cave generation. Master's thesis, IT University of Copenhagen (2014)

19. Martin, M.: Finding Next Gen – CryEngine 2. Crytek GmbH (2007)

20. Musgrave, F.K., Kolb, C.E., Mace, R.S.: The synthesis and rendering of eroded fractal terrains. Computer Graphics 23(3), 41–50 (1989)

21. Olsen, J.: Realtime procedural terrain generation. Tech. rep., University of Southern Denmark (2004)

22. van Lawick van Pabst, Jense, H.: Dynamic terrain generation based on multifractal techniques. In: T.P. Chen M., J.A. Vince (eds.) High Performance Computing for Computer Graphics and Visualisation, pp. 186–203. Springer London, London (1996)

23. Parberry, I.: Designer worlds: Procedural generation of infinite terrain from real-world elevation data. Journal of Computer Graphics Techniques 3(1), 74–85 (2014)

24. Sanzana, P., Gironas, J., Braud, I., Branger, F.: A gis-based urban and peri-urban landscape representation toolbox for hydrological distributed modelling. Modelling & Software 91, 168–185 (2017)

25. Smelik, R., Tutenel, T., De Kraker, K.J., Bidarra, R.: Integrating procedural generation and manual editing of virtual worlds. In: Proceedings of the 2010 workshop on procedural content generation in games. ACM (2010)

26. Wang, Z., Qin, Z.: Application of the virtual landscape architecture geometric modeling pp. 2803–2805 (2010). DOI 10.1109/BMEI.2010.5640566

27. Zakhozhay, O.V., del Burgo, C., Zakhozhay, V.A.: Geometry of highly inclined protoplanetary disks. Advances in Astronomy and Space Physics 5(1), 33–38 (2015). DOI 10.17721/2227-1481.5.33-38