JOT: A Modular Multi-purpose Minimalistic Massively Multiplayer Online Game Engine

Gonçalo N. P. Amador
Instituto de Telecomunicações
Covilhã, Portugal
g.n.p.amador@gmail.com

Abel J. P. Gomes
Universidade da Beira Interior
Covilhã, Portugal
agomes@di.ubi.pt

ABSTRACT

Most game engines are developed by and to the game industry. They are normally *monolithic*, i.e., typically grown out of a specific game, tendentiously gender specific, whose components (e.g., physics sub-engine) cannot be considered as separate modules, i.e., their functionalities are not clearly separated from each other. Consequently, there is a lack of *modular*, well documented, open source frameworks for games. In this paper, we present JOT, a minimal, modular game engine for Massively Multi-player Online Games (MMOGs); recall that 'jot' means minimal thing.

Author Keywords

Game Engine; Educational Software Tools; MMOGs.

INTRODUCTION

In game industry, game engines are used to produce games. Their goal is to reduce game development time through a collaborative process between programmers and artists. Game engines allow that both programmers and artists work separately or simultaneously in a given project. Historically, game engines evolved into two major variants, namely: *monolithic* and *modular*. *Monolithic* game engines evolved from specific games in order to allow content and behavior changes, via scripting (e.g., Unreal Engine [30]). *Modular* game engines are a set of tools for either game gender that, in theory, give freedom to alter the source code or replace or modify its core components (e.g., render sub-engine, physics sub-engine, AI sub-engine, etc.). Other engines (e.g., Unity3D [6]), built for multi-cross platform purposes, are in fact game engines to built *monolithic* game engines [21]. Regardless, former core sub-engines can not be replaced; instead, they can be extended with new features, i.e., they are partially *modular* game engines [21]. The main contribution of this paper is JOT, a modular game engine for research and teaching of game engine technologies and MMOGs [1].

The remain of this paper is organized as follows. Section 2 overviews game engines. Section 3 briefly describes JOT architecture. Section 4 details JOT *infrastructure layer*. Section 5 details JOT *core layer*. Section 6 details JOT *toolkits layer*. Section 7 details JOT *framework layer*. Finally, Section 8 draws relevant conclusions and points out new directions for future work.

GAME ENGINES OVERVIEW

Game Engines: the Past

The first game engines appeared in the 1980's (e.g., ASCII's RPG Maker [11]), in particular for 2D games. In the 1990's, with the launch of general purpose application programming interface (APIs), for rendering 2D and 3D vector graphics (i.e., OpenGL and DirectX), game developers began using higher level languages. More specifically, in the 1993, id Software "Doom engine" became the first commercially available 3D game engine [4, 26]. It was the first engine to structure the components of computer games (e.g., rendering, A.I., physics, networking, etc.). This allows for using the same game engine to develop other games with other contents. Regardless, until the end of the 1990's game engines where mostly focused on a specific game genre (e.g., First-Person Shooters (FPS)) [4, 26].

After 2000, game engines development went in various directions, namely:

1. Multiple game genders and platforms, e.g., Unity3D [6].
2. Mobile devices, e.g., wGEM [28], M3GE [15], mOGE [19], etc.
3. Teaching/studying game technologies, e.g., Gedi [8], Minueto [9], enJine [23], Mammoth [17], etc.

Videojogos'16, Nov 24–25, 2016, Covilhã, Portugal
Frutuoso Silva e Pedro Santos (eds.) ISBN 978-989-20-7148-0.

4. Design of novel game engines from scratch with arguably novel architectures, e.g., CryEngine [22] or Amazon Lumberyard.

5. Porting of game engines components (e.g., render module) functionalities into the GPU, e.g., via shader languages [18].

Game Engines: the Present

At the present, game engines are in their vast majority complex tools, often permitting to recode portions of the engine core or the modify it via a general (e.g., Lua) or specific (e.g., UnrealScripting) scripting language. Some game engines strive for being suited for multiple genders and multiple platforms, e.g., Unity3D [6]. Note that many game engines developed in academia for a specific field of study are either no longer open source [13, 23, 3, 7, 16, 17] or abandoned [9]. Nevertheless, open source game engines are to complex or not enough modular, so that the removal or rewriting of its components in a difficult task. But, recall that most major game companies use proprietary game engines.

Game Engines: the Future

We have identified five major future research trends in the development of game engines, namely:

1. The replacement of traditional raster-based renderers by ray tracing-based renderers [31]. In the render module of game engines and other graphical applications, ray casting (i.e., a particular ray tracer limited to primary rays) has been used in game development for many years [12], mainly for collision detection. This is so because because it is already feasible to run a ray tracer in real-time for no so much complex 3D scenes using shader programming or more general programming like CUDA or OpenCL, so that the expectations are to have ray tracers with real-time performance in the near future [25].

2. The introduction of voxelized scenes for scene management, including physics and complex terrains. This particularly useful for handling terrain features like caves or overhangs, as well as destructive war scenarios with falling off buildings and the like.

3. The development of automated procedural generation algorithms. One of the major problems of current games is the construction of scenarios for game levels, though most game engines include scene editors. In the future, we will see automated generation of 3D scenes from a a descriptive specification in order to shorten the process of game prototyping and development.

4. GPU-centric game engines. At present, most parallel features of game engines are provided by shaders, particularly for illumination purposes, though some efforts were made to integrate commercial games with NVidia CUDA (e.g., Mirror's Edge [24]). Therefore, in the future, we will assist to the development of the game engines that fully leverage the processing power of GPUs.

5. The emergence of really modular game engines. Nowadays, most game engines do not permit the replacement of a game core component easily. In fact, there is no known game engine or framework that allows for the replacement of any sub-engine (e.g., physics, A.I., networking, etc.) by another one; for example, it is not possible to replace a physics sub-engine (e.g., Havok) by another one (e.g., Bullet) in a straightforward manner. It is clear that this would be feasible if there was a common interoperability framework for physics sub-engines.

JOT architecture takes into consideration the former aspects to attempt to ensure modularity and future extensibility.

JOT ARCHITECTURE

JOT is built in Java programming language for several reasons, namely:

1. The Java Virtual Machine (JVM) is multiplatform.

2. There are considerable third-party Java application layer multicast implementations (e.g., JGroups) and cloud/grid middlewares (e.g., GridGain).

3. There is a significant base of game engines for teaching computer graphics courses and/or studying game technologies in Java, e.g., M3GE [15], Minueto [9], enJine [23], Mammoth [17], etc.

4. There are open source game engines in Java that serve as a base for design decisions and performance testing, e.g., jMonkeyEngine and Jake2, an open source port to Java of Quake II.

JOT gathers many concepts of other game engines developed in academia. First, it has a layer structure similar to that one of RAGE [20]. Second, it uses concepts implemented in M3GE and jMonkeyEngine scene management, i.e., its classes and methods are similar to those found in M3GE and jMonkeyEngine. Third, the organization of toolkits of JOT is strongly influenced by Mammoth; e.g., JOT *artificial intelligence toolkit* uses a similar design as Mammoth AI module. JOT is a four-layer game engine, from bottom to top: *Infrastructure*, *Core*, *Toolkits*, and *Framework*, as illustrated in Fig. 1.

As observed in Fig. 1, dashed rounded corner rectangles are external libraries, e.g., *JOGL*. Any layer component that resorts to an external library obeys to interface/abstract classes as expected by above layers, e.g., if we replace JGroups with Orbit, solely the *Network Toolkit* must be re-implemented obeying to the same interface/abstract classes, without any changes required in other toolkits or layers. Each layer constituents can only use functionalities of the same layer or lower layers constituents, e.g., the *Artificial Intelligence Toolkit* relies on the *Geometry Extended Toolkit*, on the *Core layer*, and on the *Infrastructure layer*. In order to replace a component in the engine, one might solely adapt the respective toolkit or additionally modify its lower layers, e.g., to replace JOT physics code with a physics engine one must implement classes that obey to the provided interface/abstract classes in the *Physics Extended Toolkit* and, if desired, in physics *Core* layer. This implies that modularity of the game engine is ensured at the expense of possible redundancy or unused source code in the *Core layer*.

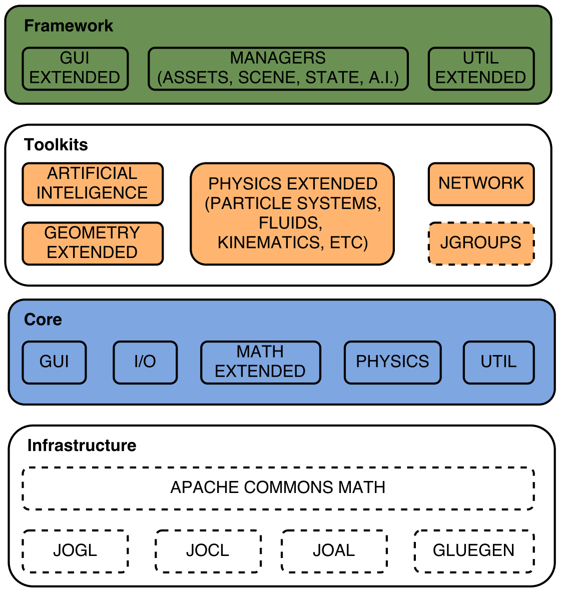

Figure 1. JOT layers.

INFRASTRUCTURE LAYER
This layer is comprised of five libraries: *The Apache Commons Mathematics Library*, *JOGL*, *JOCL*, *JOAL*, and *GLUEGEN*, with the latter four wrapped in *JogAmp*. Note that JogAmp provides the graphical pipeline, sound, and OpenCL support. Presently, JOT uses JOGL for the graphical pipeline (OpenGL) and for 2D/3D sound support (OpenAL). Recall that there is not a Java binding for DirectX. *The Apache Commons Mathematics Library* is an extensible math library for general purpose scientific use, chosen by its extensive features and continuous support/development, in detriment of Java3D Vecmath, used for example in jMonkeyEngine. However, many of the *The Apache Commons Mathematics Library* functionalities are implemented for doubles and not for floats, thus some upper layers of JOT re-implement some of its functionalities for single precision floating-point numbers instead. More libraries, namely, OpenCV for camera or control devices, such as the WiiMote, might also be added in this layer.

CORE LAYER
Core is the minimal set of tools that can allow the classification of a piece of software as a game engine, i.e., games can be written on the top of the *Core* layer, with the assumption that we have the infrastructure layer underneath. The *Core* layer is comprised of five modules, as detailed below.

GUI Module
This module regards a simple game graphical application with the following functionalities: initialize the game, i.e., load the menus/controls devices and show startup info; load all default game content, i.e., player, obstacles, SkyBox, and floor; running the game main thread; shutdown the game, clean up code that runs only once; the loop that runs until the game ends, that continuously and as many times as possible per second processes the input handlers events, update the state and position of all the game objects, detect collisions and provide responses, and refreshes the display. Some of this module abstract single class *SimpleGame.java* methods are guidelines (abstract methods) to be implemented in a game project, i.e., an game main class extends this class.

Math Extended Module
This module is built upon *The Apache Commons Mathematics Library*. It regards geometry in a graphical scene. It contains *shape* class and its subclasses for triangles, planes, and spheres, which are the constituents of a *mesh* class. Note that each mesh object (and its bounding volume(s)) can be associated to a *transform group* for scene modeling purposes, as usual in scene graphs. Implemented *bounding volumes* are dynamic axis aligned bounding box (*AABB*), object aligned bounding box (*OBB*), and *bounding sphere*. A dynamic AABB changes size depending on the mesh rotation to ensure that all mesh vertices are within the AABB. Obviously, the abstract classes for shapes and bounding volumes are written to accommodate new subclasses for specific shapes and bounding volumes, i.e., they can be extended. Also, this module implements single and double precision real numbers, linear and cubic interpolation, and geometric intersection algorithms, as needed in collision detection, though only the Gilbert-Johnson-Keerthi (GJK) distance algorithm has been implemented [14].

I/O Module
This module is where the 2D/3D sound, scripting engine, control devices, and geometric loaders are implemented. The geometric loaders currently support the Collada 1.4.1 format (i.e., geometry and material data), but Wavefront OBJ and MD2 quake 2 files formats are not fully supported. The sound is managed by an handler for 2D/3D sound, which was implemented on top of OpenAL. The scripting engine serves the purpose of loading configuration options in the *Util* module (more specifically, in*CoreOptions* class), though this scripting engine might be extended to incorporate other functionalities. Note that the I/O module is designed in order to be extended with classes or libraries to interpret scripting languages (e.g., Lua or Phyton), and other control devices aside mouse or keyboard, such as WiiMote or Kinnect. The control devices classes implement one or two interfaces, namely:

1. *Input* is the interface for all control devices that do not require their positions to be tracked, e.g., keyboard, joystick, steering wheel, WebCam, Kinect, etc.

2. *TrackableInput* is the interface for all control devices that require tracking of their positions or acceleration, e.g., mouse, Wiimote, etc.

These two interfaces require the registration of user-interaction events, e.g., a key or button is pressed in a specific position. This is achieved using the *isDetecting* or *isContinuouslyDetecting* methods. Trackable controls are those that have gyroscopes and/or accelerometers.

Physics Module

The physics module consists of the following interfaces/classes: *material* (for objects), *ray* (of light), *intersectionResult* (between ray and objects). Note that the *mesh* class has a list of materials. The *ray* and *intersectionResult* classes are for upper layer support for ray casting [31] or ray tracing [12] implementations.

Utils Module

This module specifies the options (boolean values) of the core, and implements a generic class called *CoreGameObject*. The *GameObject* class is an extension to the abstract class *CoreGameObject*, which in turn extends the *TransformGroup* class. A *GameObject* is the most generic type of game object, no matter the the game gender; this class includes not only data concerning geometry and materials, but also data related with speed, maximum speed, attributes, and so on.

TOOLKITS LAYER

This layer includes toolkits, which are extensions to the core. There are toolkits for a number of purposes, namely artificial intelligence (AI), geometry generators, physics simulation, etc. Each toolkit includes an *Utils* component which specifies its options (boolean values).

Geometry Extended Toolkit

This toolkit extends the features of core geometry module, with: *terrain generators*, via fractal or Brownian motion noise generators; *terrain smoothing* algorithms, via parametric surfaces or Gaussian blur/smooth; *maze generation*, using Prim's algorithm; *quadrics*, *SkyDome*, and roughly spherical *celestial objects* (planets, moons, and stars).

Physics Extended Toolkit

This toolkit provides several new features to the core physics, namely: *collision handling* among rigid (either moving or not) bodies; *kinematics* for moving bodies and for projectiles; *particle system effects*, for explosions and 2D fluids; *Euler 2D fluids*; *hybrid* (Euler and particle fluid physics) or *procedural* rain ripply effects. As a margin note, work is undergoing to extend all these 2D physics effects to 3D.

Artificial Intelligence (AI) Toolkit

The *AI Toolkit* supports steering behaviors [29], where each behavior implements a common *steering behavior* interface. The implemented steering behaviors are, *seek*, *pursuit*, *flee*, *evade*, *arrive*, *collision avoidance*, *interpose*, *wander*, *path following*, *offset pursuit*, and *flocking behavior*. It also supports pathfinding algorithms, where each pathfinder implements a *pathfinder* interface. The implemented pathfinders include Dijkstra's algorithm [10], A* [2], Fringe search [5], and Best-First search [27].

Network Toolkit

This toolkit implements the generic network communication of JOT. It implements an interface that serves to abstract communication using any networking API (e.g., JGroups and GridGain), regardless of its network topology. Therefore, if the networking API was replaced by another for some reason, game implementations that resort to this toolkit will still work.

FRAMEWORK LAYER

This is the upper layer of JOT. This layer comprises three modules: GUI, managers, and Util Extended. This layer aims at the following: first, to provide management of the application/game state and scene; second, to separate the game logic from its graphical application. As a margin note, a partially implemented off-line ray-tracer is integrated within this layer.

GUI Extended Module

This module provides the following classes: *FramePerSecond*, *Game* (an abstract class that every single game must extend), and *Camera* (another abstract class). This latter class has the following sub-classes: *FirstPerson* for first-person shooters, *ThirdPerson* for role-playing games, and *StaticPerspective*, *TrackingGameObjectPerspective*, *StaticUpperView* and *TrackingGameObjectUpperView* for strategy games.

Manager Module

This module is responsible for scene, state, and AI managers are implemented. The scene manager deals with handling of the scene via a tree-based scene graph, permits the usage of planar or shadow volume-based shadows, and handles collisions among objects within the scene graph. The state manager handles game state, i.e., information of each game object (e.g., life, speed, etc.), no matter we are using a client-server network topology or a peer-to-peer network topology; which is particularly useful for massively multiplayer online games (MMOGs). Finally, there are two AI managers: the first for steering behaviors and the second for path-finding.

Util Extended Module

This module implements a specification of options (boolean values) of the *Framework* layer for usage in the game application/implementation.

CONCLUSIONS AND FUTURE WORK

JOT current architecture is satisfactory in proposed features and modularity. However, JOT still lacks several features and improvements to existent features, namely:

1. Extend the *I/O* module, to fully support Wavefront OBJ and MD2 quake 2 files format, and Collada 1.4.1 or 1.5 skeleton animation.

2. Extend the Euler and particle fluids to 3D.

3. Replace JOT toolkits with third-party solutions, e.g., replace the *physics extended toolkit* with the JBullet physics engine.

4. Improve JOT in order to make it a common interoperability framework or interface among the many game related middlewares for many tasks, physics and render to name a few.

5. Paralellize JOT in order to make it a general purpose parallel game engine, using OpenCL. To be clear, we do not refer solely to using shaders mostly for illumination effect and/or render. Instead the idea it to run most of the game engine features inside the graphics card, e.g., a Ray Caster [12], e.g., for collision detection, and a Ray Tracer [31] for rendering of the game.

6. Improve JOT in order to make it a common interoperability framework or interface among the many game related middlewares for many tasks, physics and render to name a few.
7. Support JOT with cloud technologies, i.e., make it able to be used by multiple worldwide distributed game developers.
8. Allow games to be rendered either directly on the browser or as an application.
9. Modify JOT in order to allow it to be used in the creation of mobile applications for Android.

As priorly stated a few of these improvements are already undergoing work.

ACKNOWLEDGMENTS

This research has been partially supported by the Portuguese Research Council (Fundação para a Ciência e Tecnologia), under the doctoral Grant SFHR/BD/86533/2012, and also by FCT Project UID/EEA/50008/2013.

REFERENCES

1. G. Amador and A. Gomes. 2016. A Video Games Technologies Course: Teaching, Learning, and Research. In *EG 2016 - Education Papers*. The Eurographics Association.
2. A. Bagchi and A. Mahanti. 1983. Search Algorithms Under Different Kinds of Heuristics–A Comparative Study. *J. ACM* 30, 1 (1983), 1–21.
3. J. Bernardes, R. Nakamura, D. Calife, D. Tokunaga, and R. Tori. 2009. Integrating the Wii Controller with enJine: 3D Interfaces Extending the Frontiers of a Didactic Game Engine. *Computers in Entertainment (CIE)* 7, 1 (2009), 1–19.
4. L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz. 1998. Designing a PC Game Engine. *IEEE Computer Graphics and Applications* 18, 1 (1998), 46–53.
5. Y. Björnsson, M. Enzenberger, R. Holte, and J. Schaeffer. 2005. Fringe Search: Beating A* at Pathfinding on Game Maps. In *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG '05), Essex University, Colchester, Essex, UK, 4-6 April, 2005*. IEEE Computer Society.
6. S. Blackman. 2011. *Beginning 3D Game Development with Unity: All-in-one, Multi-platform Game Development* (1st ed.). Apress, Berkely, CA, USA.
7. Jean-Sébastien Boulanger. 2006. *Interest Management For Massively Multiplayer Games*. Master's thesis. School Of Computer Science, McGill University, Montréal, Canada.
8. R. Coleman, S. Roebke, and L. Grayson. 2005. Gedi: a game engine for teaching videogame design and programming. *Journal of Computing Sciences in Colleges* 21, 2 (2005), 72–82.
9. A. Denault. 2005. *Minueto, an Undergraduate Teaching Development Framework*. Master's thesis. School Of Computer Science, McGill University, Montréal, Canada.
10. E. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *NUMERISCHE MATHEMATIK* 1, 1 (1959), 269–271.
11. M. Duggan. 2011. *RPG Maker for Teens* (1st ed.). Course Technology Press, Boston, MA, United States.
12. J. Ellis, G. Kedem, T. Lyerly, D. Thielman, R. Marisa, J. Menon, and H. Voelcker. 1991. The Ray Casting Engine and Ray Representatives. In *Proceedings of the First ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications (SMA '91)*. ACM Press, 255–267.
13. L. Emmerich, D. Tanaka, R. Petriche, F. Kamakura, and J. Bernardes. 2006. Building the Network Module for a Didactic Game Engine. (2006).
14. E. Gilbert, D. Johnson, and S. Keerthi. 1988. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation* 4, 2 (1988), 193–203.
15. P. Gomes and V. Pamplona. 2005. M3GE: um motor de jogos 3D para dispositivos móveis com suporte a Mobile 3D Graphics API. In *Proceedings of the IV Brazilian Symposium on Computer Games and Electronic Entertainment*. Sociedade Brasileira de Computação (SBC), 55–65.
16. M. Hawker. 2008. *Subgames in massively multiplayer online games*. Master's thesis. School Of Computer Science, McGill University, Montréal, Canada.
17. J. Kienzle, C. Verbrugge, B. Kemme, A. Denault, and M. Hawker. 2009. Mammoth: a massively multiplayer game research framework. In *Proceedings of the 4th International Conference on Foundations of Digital Games (FDG '09)*. ACM Press, 308–315.
18. A. Lefohn, M. Houston, J. Andersson, U. Assarsson, C. Everitt, K. Fatahalian, T. Foley, J. Hensley, P. Lalonde, and D. Luebke. 2009. Beyond Programmable Shading (Parts I and II). In *ACM SIGGRAPH 2009 Courses*. ACM Press, 1–312.
19. I. Macêdo, Jr. 2005. *mOGE - mObile Graphics Engine: O projeto de um motor gráfico 3d para a criação de jogos em dispositivos móveis*. Final report. Universidade Federal de Pernambuco (UFPE), Pernambuco, Brazil.
20. L. McCulloch, A. Hofman, J. Tulip, and M. Antolovich. 2005. RAGE: A Multiplatform Game Engine. In *Proceedings of the Second Australasian Conference on Interactive Entertainment (IE '05)*. Creativity & Cognition Studios Press, 129–131.
21. F. Messaoudi, G. Simon, and A. Ksentini. 2015. Dissecting Games Engines: The Case of Unity3D. In *Proceedings of the 2015 International Workshop on Network and Systems Support for Games (NetGames '15)*. IEEE Press, 4:1–4:6.

22. M. Mittring. 2007. Finding Next Gen: CryEngine 2. In *ACM SIGGRAPH 2007 Courses*. ACM Press, 97–121.

23. R. Nakamura, L. Bernardes, and R. Tori. 2006. enJine: Architecture and application of an open-source didactic game engine. In *Proceedings of the Digital V Brazilian Symposium on Computer Games and Digital Entertainment (SBGAMES'2006)*.

24. J. Norton, C. Wingrave, and J. LaViola, Jr. 2010. Exploring Strategies and Guidelines for Developing Full Body Video Game Interfaces. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games (FDG '10)*. ACM Press, 155–162.

25. S. Parker, H. Friedrich, D. Luebke, K. Morley, J. Bigler, J. Hoberock, D. McAllister, A. Robison, A. Dietrich, G. Humphreys, M. McGuire, and M. Stich. 2013. GPU Ray Tracing. *Commun. ACM* 56, 5 (2013), 93–101.

26. P. Paul, S. Goon, and A. Bhattacharya. 2012. HISTORY AND COMPARATIVE STUDY OF MODERN GAME ENGINES. *International Journal of Advanced Computer and Mathematical Sciences* 3, 2 (2012), 245–249.

27. J. Pearl. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

28. A. Pessoa, G. Ramalho, and A. Battaiola. 2002. wGEM : Um Framework de Desenvolvimento de Jogos para Dispositivos Móveis. In *Proceedings of the XXIX Seminário Integrado de Software e Hardware*. Sociedade Brasileira de Computação (SBC).

29. C. Reynolds. 1999. Steering Behaviors For Autonomous Characters. (1999). Retrieved September 16, 2016 from `http://www.red3d.com/cwr/papers/1999/gdc99steer.pdf`.

30. L. Surhone, M. Tennoe, and S. Henssonow. 2010. *UnrealScript*. Betascript Publishing, Mauritius.

31. T. Whitted. 1980. An Improved Illumination Model for Shaded Display. *Commun. ACM* 23, 6 (1980), 343–349.