electronics

MDPI

Article

Developing an Open-Source Lightweight Game Engine with DNN Support

Haechan Park and Nakhoon Baek *,†

School of Computer Science and Engineering, Kyungpook National University, Daegu 41566, Korea; lucidsun92@gmail.com

* Correspondence: nbaek@knu.ac.kr; Tel.: +82-53-950-6379

† Currently visiting: School of Electrical Engineering & Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA.

Received: 15 June 2020; Accepted: 13 July 2020; Published: 1 September 2020

Abstract: With the growth of artificial intelligence and deep learning technology, we have many active research works to apply the related techniques in various fields. To test and apply the latest machine learning techniques in gaming, it will be very useful to have a light-weight game engine for quick prototyping. Our game engine is implemented in a cost-effective way, in comparison to well-known commercial proprietary game engines, by utilizing open source products. Due to its simple internal architecture, our game engine is especially beneficial for modifying and reviewing the new functions through quick and repetitive tests. In addition, the game engine has a DNN (deep neural network) module, with which the proposed game engine can apply deep learning techniques to the game features, through applying deep learning algorithms in real-time. Our DNN module uses a simple C++ function interface, rather than additional programming languages and/or scripts. This simplicity enables us to apply machine learning techniques more efficiently and casually to the game applications. We also found some technical issues during our development with open sources. These issues mostly occurred while integrating various open source products into a single game engine. We present details of these technical issues and our solutions.

Keywords: game engine; open source; artificial intelligence; light-weight; deep neural network; implementation; case study

1. Introduction

Recently, it is emerging to apply artificial intelligence and deep learning techniques to various fields [1–3]. Accordingly, the game industry is also using cases of artificial intelligence and deep learning techniques for games and/or game engines. Actually, popular game engines like the Unity engine from Unity Technologies (San Francisco, LA, USA) and the Unreal engine from Epic Games (Cary, NC, USA) provide machine-learning modules. In the case of the Unity engine, an machine learning (ML)-agent project is actively in progress to apply the machine learning techniques into the game development. Researchers represent their studies on the Unity blog [4], and projects are published on their GitHub pages [5]. Similarly, the Unreal engine supports a machine learning plug-in [6].

However, these commercial proprietary game engines are not suitable for light-weight uses, such as the simple prototype testing purpose. Due to the heavy functions and complex architectures of commercial proprietary engines, it is difficult to verify implementation results and/or to modify internal structures, or more precisely, the functional relationships for internal components. Analyzing large functional relationships would be time-consuming and has drawbacks to future development schedules and decisions [7].

In contrast, our game engine is implemented as a light-weight system, by using open source products [8,9]. This game engine enables relatively short schedules to release project results, due to their simple and efficient architectures. It is also effective in situations when modifying and reviewing the new functions through quick and repetitive tests. In addition, the built-in DNN (deep neural network) module operates various deep learning algorithms in real-time, simultaneously with the main game features.

However, developing a game engine with open source products introduce many technical issues. Since the game engine utilized various open source products, many conflicts occurred while integrating them. This paper represents the structure of our game engine and, for the demonstration purpose, it shows a bowling game with a neural style transfer [10] applied to texture images in the Windows 10 PC environment. Finally, this paper highlights various kinds of issues mainly occurred while integrating open source products and represents the directions of the future researches.

2. Related Works

In the software engineering field, an engine means an architecture or a framework, to provide a software development environment [11]. Usually a software engine provides various tools and collections of libraries, to support a specific program. Accordingly, a game engine is a software engine to develop game programs [12].

In 1970s, early personal computers and also game consoles had limited computing powers, and thus, most of them should be fully optimized to use the full power of their underlying devices [13]. In 1980s, the first generation video game consoles have been released. For those consoles, console makers have provided special-purpose programming languages, special-purpose compilers, and other tools, for their console-based game development [9].

In 1990s, they started to reuse the source codes of published computer games to develop additional games. The famous Quake engine and Unreal engine have their origins from the famous computer games Quake III Arena and Unreal, respectively [6].

Now, in the field of computer games, game engines are widely used, and common to computer game developers. The currently famous commercial proprietary game engines can be summarized as follows:

- Unreal engine: It is developed by Epic Games, for first-person shooting games [14]. It was initially used for the Unreal game, in 1998. It is now successfully used in various game fields, and one of the most-widely used game engines.
- Unity engine: It is a cross-platform game engine developed by Unity Technologies. This game engine now supports more than 25 platforms [15]. It has been adopted for other purposes, including special effects, architectural design, and other engineering visualizations, in addition to the original game programs.
- CryEngine: This game engine is developed by Crytek (Frankfurt, Germany), and initially used for their games [16]. Now, it is used by various games, and additionally, licensed to Amazon.com. Later, Amazon.com released the Amazon Lumberyard [17], the extended version of this CryEngine.

Recently, we have a new trend of providing light-weight game engines. These light-weight game engines are ordinarily open sourced. Generally, an open source platform has the following strong points:

- Source code availability: Many developers choose an open source game engine, mainly due to the wide availability of the source codes. They can check detailed processing, and also can modify to accelerate and/or to customize the game engine for their specific purposes.
- Expandability: Since the source code is open to the public, additional device drivers and also new plug-ins can be developed. Thus, the game engine itself can be expanded, and also can be upgraded with these new features.

- Cost-effectiveness: In most cases, open source game engines have free license policy. Of course, some open source game engines even require specific licenses. In any case, the whole cost of using an open source game engine is much inexpensive, in comparison to the commercial proprietary game engines.

We can summarize the open-source light-weight game engines as follows:

- Cocos2D: This game engine is one of the most widely used ones, which is developed in the C++ programming language. Actually, this game engine is a collection of game developing tools, including Cocos2D-objc, Cocos2D-x, Cocos2d-html5, and Cocos2D-XNA [18].
- Godot: The Godot engine [19] provides both of two-dimensional (2D) and three-dimensional (3D) game building features. This open source game engine can be used through the MIT license, without any royalties and any subscription fees. It is widely used for cross-platform game programs.
- Gdevelop: It is another open source game engine, supporting multi-platforms, including Windows, macOS, and Linux platforms, as other open source game engines [20]. It is especially good for fast prototyping.

Recently, various works have used machine learning techniques in the game development. For example, they used it for recommending useful items for players to use in the game [21], choosing smarter behavior for enemies [22], generating game assets with generative algorithms [23], and others. Most researchers have used the machine learning plug-ins of the Unity engine and/or Unreal engine [5,6]. However, in some cases, they also want light-weight game engines, because commercial proprietary game engines are too complicated to access or to modify. Some of them implemented their own light-weight game engines [8]. In this paper, we aim to implement a light-weight game engine with the DNN module to support integrating machine learning techniques into game applications.

3. Design and Implementation of a Light-Weight Game Engine

A game engine is a kind of development tool. It supports the game developers to make games more efficiently and effectively, by providing some common features for game development in advance. Some game engines are easy to create games, which can be executed on various types of platforms, while others are fit only for a particular genre and/or a specific platform. This difference is due to the fact that each game engine has different development purposes. Our proposed game engine is actually designed to integrate various modules, and to create 3D games in the Windows 10 PC platforms, with the C++ programming language.

Our game engine is light-weight: it means each module and the entire structure of the game engine is implemented in an intuitive and efficient way. The main purpose of our game engine is to integrate various open sources and build up an agile system. Many modules of our game engine are inherited from open source projects, and tuned for rapid prototyping, through the iterative development.

The major components of a typical game engine would be the rendering engine to represent the virtual game world onto the screen, the audio engine to produce music and sound effects, the physics engine to apply natural motion effects between objects, the event system to handle various events that occur on the screen and other devices, the game logic system, and others. Additionally, the network system, the database system and the animation system can be additionally integrated into the game engine [9].

Figure 1 represents the overall structure of our game engine. From the application programmer's point of view, the OnStart function is provided to perform initialization tasks to be executed only once. The following Run function manages the main game loop. When the game receives an end event (for example, when the user closed the application window), the loop exits and the game ends. Inside the game loop, the game status values to be updated are calculated for each frame, and these updated values are applied to the next frame on the screen or other devices accessed by the modules.

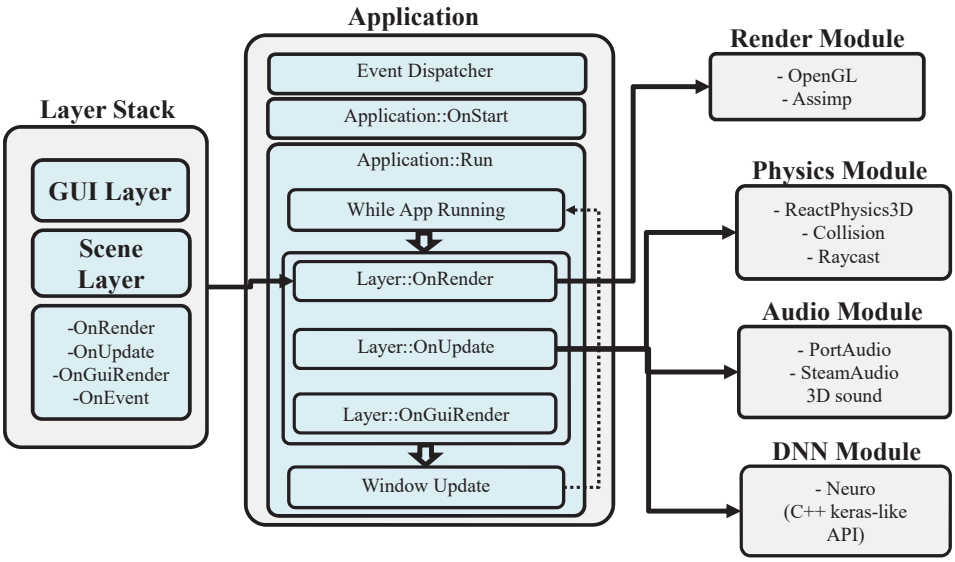


Figure 1. The overall architecture of our game engine.

The sub-functions and modules executed in the Run function can be described as follows:

- The OnRender function draws game objects on the screen using a rendering module implemented with OpenGL (Open Graphics Library) [24]. Drawing member functions of all 3D objects are called in this function.
- The OnUpdate function performs calculations for tasks, changing over time except for rendering. In the game engine, the OnUpdate function accumulates the playtime and sends the time spent for the frame, to the physics module, which updates the physical quantities for the next frame.
- The OnGuiRender function performs user interface (UI) renderings on the screen. We used the OpenGL implementation of an open source C++ graphical user interface tool, named ImGui [25]. Using ImGui with docking functions, ImGui UI windows can be docked and moved over each other.

The Layer Stack is a stack of layers: each layer has its own member functions including OnRender, OnUpdate, OnGuiRender, and OnEvent. We separated the layers according to the management purposes. As an example, a mouse click on a UI button will be processed by the UI layer, and will be propagated to the underlying motion layer. Without the UI layer, the underlying motion layer may get the mouse clicks as triggering actions or other action events. In some cases, putting all the game objects into a single layer may cause the game logic algorithms to be over-complicated. Additionally, separated layers are convenient for the resource managements: removing unnecessary layers from the LayerStack is an efficient way of maintaining memory space adequately. More details on our modules will be described in the following subsections.

3.1. Rendering Module

The rendering module is mainly implemented with the 3D graphics library of OpenGL [24,26]. This module conducts various matrix calculations to place 3D objects on the screen. It also uses shader programs and frame buffers to create various rendering effects and post-processing effects. The model loader draws 3D graphics models generated by 3D modeling tools and also the output primitives of the game engine. To efficiently upload and use the 3D graphics models, we use the Open Asset Import Library (Assimp) [27].

Figure 2 shows the internal structure of our rendering module. All game objects are declared in the SceneData area of the SceneLayer. In the SceneData area, draw methods of each game object are registered to be executed in the OnRender function. To initialize and update the game world in the SceneData area, we need the positions and orientations of lights and cameras.

The cameras and the lights are implemented with OpenGL mathematics library (GLM) [28]. Real-time game scene updates are applied to the objects in the scene layer.

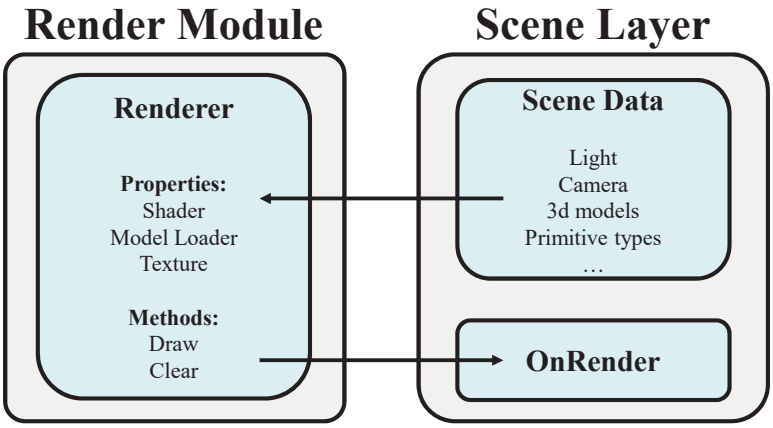


Figure 2. The rendering module.

3.2. Physics Module

The physics module is implemented with an open-source physics library React-Physics3D [29]. The physics effects are commonly applied for natural and realistic motions. Natural motion examples include post-crash movements and acceleration movements due to frictions or gravity. Figure 3 shows the structure of the physics module: the Dynamics World class acts as a controller. The member function of the Dynamics World, which is named as CreateRigidBody, is called to create a rigid body for an object and returns a pointer to the rigid body of the object. This pointer is stored in a rigid body pointer array, named as RigidBodies and a specific rigid body can be accessed by its index. A rigid body will have its own shape, and also its geometric information, including positions and orientations.

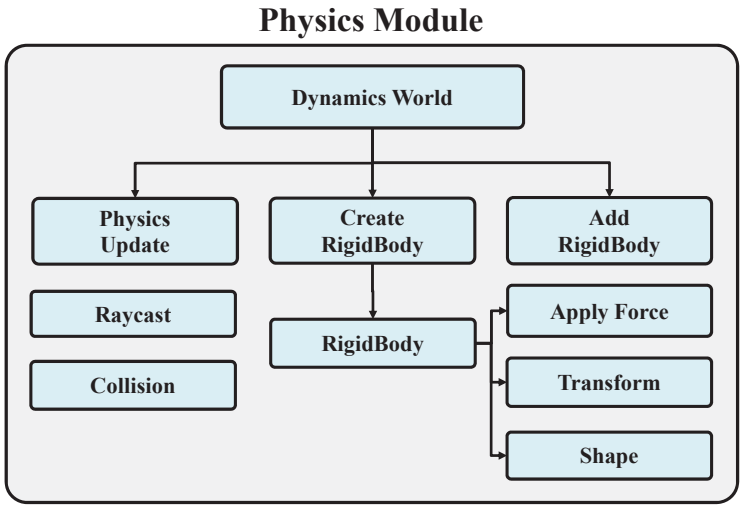


Figure 3. The physics module.

A rigid body has the ApplyForce method to apply a direct force on it. Additionally, the collision forces between rigid bodies are processed at one time, when the Dynamics World class executes the PhysicsUpdate function. For efficiency reasons, physics calculations are performed after user-provided time intervals, rather than for each rendering frames. In some cases, external forces can be applied directly to specific rigid bodies. The Raycast function provides the way to interact directly with objects on the screen. The ray starts from the center of the camera position and shoots a ray at the point of the mouse click position. Then the system will detect objects intersecting the ray.

The transformation (or equivalently, translation and rotation) matrices are calculated by the GetOpenGLmatrix function, for each game object. It returns a matrix used for rendering in OpenGL shader programs.

3.3. Audio Module

The audio module is implemented with the open source audio API (application programming interface) of PortAudio [30]. This module manages the background music and sound effects. Figure 4 is the structure of the audio module.

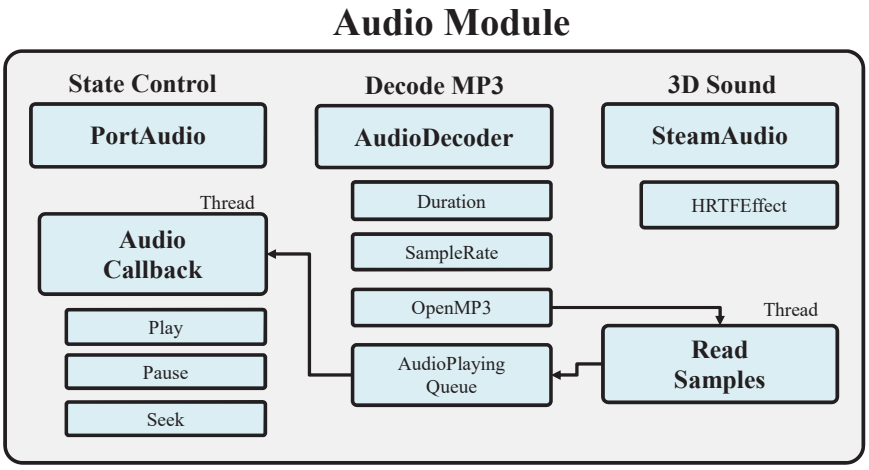


Figure 4. The audio module.

The audio module invokes AudioCallback function periodically. The AudioCallback function reads the audio data and sends it to output devices. We also integrated an open source MP3 audio decoder [31]. This audio decoder parses metadata such as the duration, the sample rate, and the bit-rate of an MP3 file. At the end of the callback function, it compares the current playback time and the duration time to decide whether it should invoke the next callback or finish the music playing.

The audio module additionally provides 3D acoustic effects by integrating SteamAudio [32]. SteamAudio modifies fetched audio data with geometric information of game objects. Moreover, SteamAudio can perform stereo sound effects for two-channel speakers. Due to this integration, users can get realistic sounds.

From the simplicity point of view, it is the easiest case when all the audio samples are completely loaded onto the main memory area. However, due to the limited memory resources, we should manage the real-time loading of the sound effect files and also the partial load of the large background music files. Therefore, our proposed game engine uses a set of threads to read audio samples from MP3 files and push the appropriate amount of samples into the audio playing queue. Then the AudioCallback reads audio data from the audio playing queue and sends them to output devices to play sounds. Since multiple access to the same audio playing queue can cause unexpected exceptional cases, we introduce mutually exclusive locking methods to the audio playing queue.

3.4. DNN Module

The DNN module is based on a C++/C# neural network implementation, which is called as Neuro library [33]. The Neuro library provides a set of API functions, which are similar to Keras in TensorFlow. Figure 5 shows the internal structure of our DNN module. With this module, many deep learning algorithms are possibly applied to game applications.

Furthermore, this module can load sufficiently trained models, and use them immediately. As an example, the weights of a deep learning model stored in the HDF5 (hierarchical data format version 5) foramt can be loaded, skipping the training process, and the trained weight values can be

used immediately. The proposed DNN module supports graphics processing unit (GPU) operations, using compute unified device architecture (CUDA), to accelerate DNN operations. The proposed module uses a separate thread to conduct DNN operations simultaneously to the game.

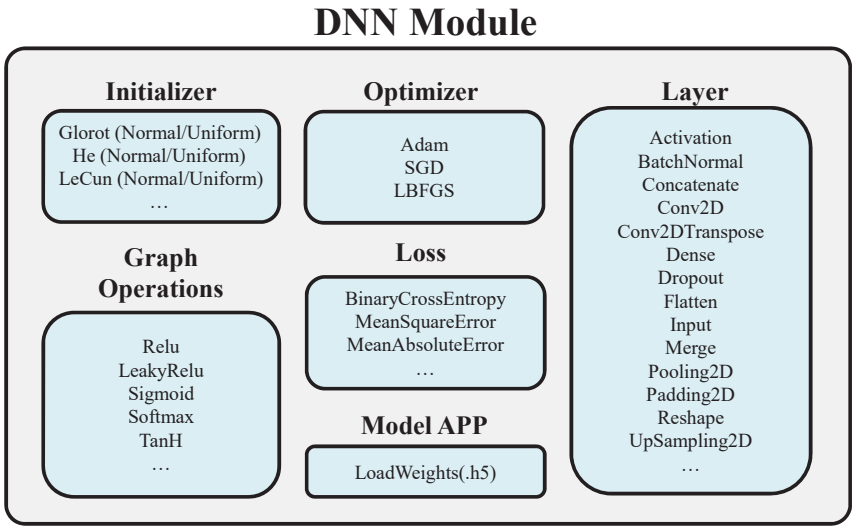


Figure 5. The DNN (deep neural network) module.

4. A Case Study

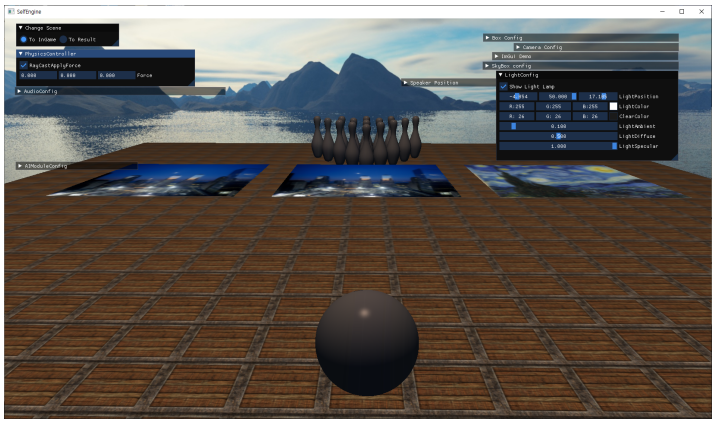
To test all the modules in the game engine, we developed a simple bowling game, with the neural style transfer [10] feature on the texture images. With this demonstration game, the physics effect can be tested by the collisions between the bowling ball and the bowling pins. The audio effects can be tested by playing a music in the game world with a 3D speaker model.

Our game environment is summarized in Table 1. In this environment, the demonstration game interactively works in real time. It shows at least more than 60 frames per second, with the smooth physical simulation and full screen 3D rendering. The neural style transfer method also works smoothly. This result shows that our game engine works correctly and also efficiently.

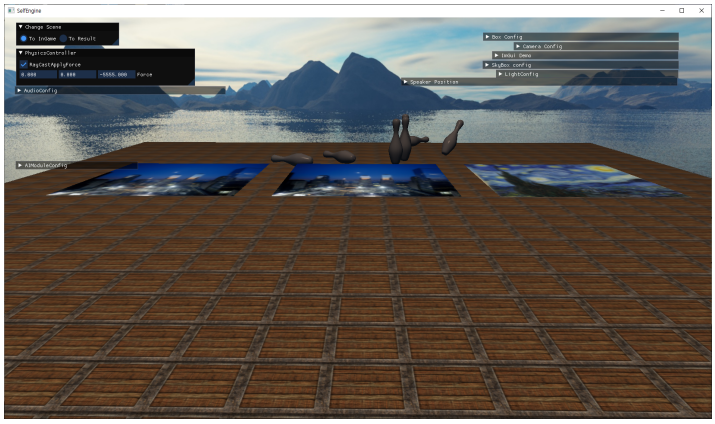
Table 1. Game engine driven device environment.

Device	Specification
CPU	Intel Core i5-4590
GPU	NVDIA GeForce RTX 2070 Super
RAM	24 GByte
OS	Windows 10 Professional Edition

Figure 6a,b are screen shots from the bowling game, before and after a collision. Figure 7a represents the beginning of the neural style transfer and Figure 7b represents the completed neural style transfer. On the screens of Figure 7a,b, there are three textures on the planar surface. The left texture image shows the original content image, the right texture image is the style image, and the center texture image shows the style transfer result. The center texture image is modified for every 50 epochs, during the training of the neural style transfer method.

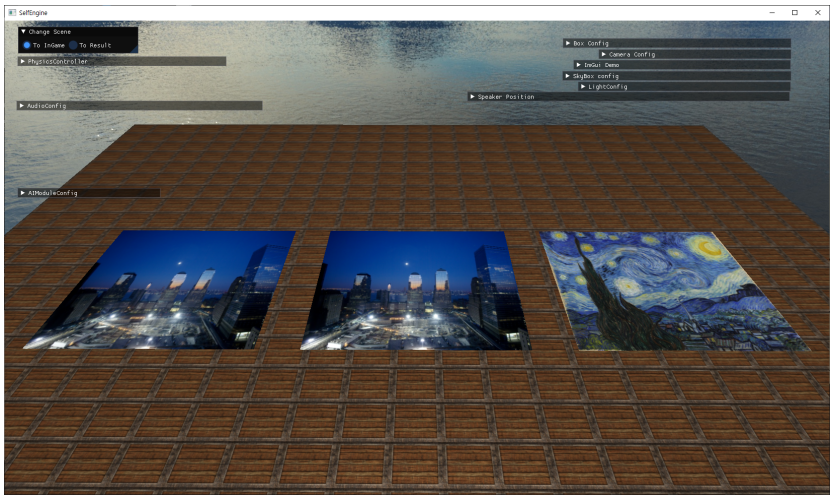


(**a**)



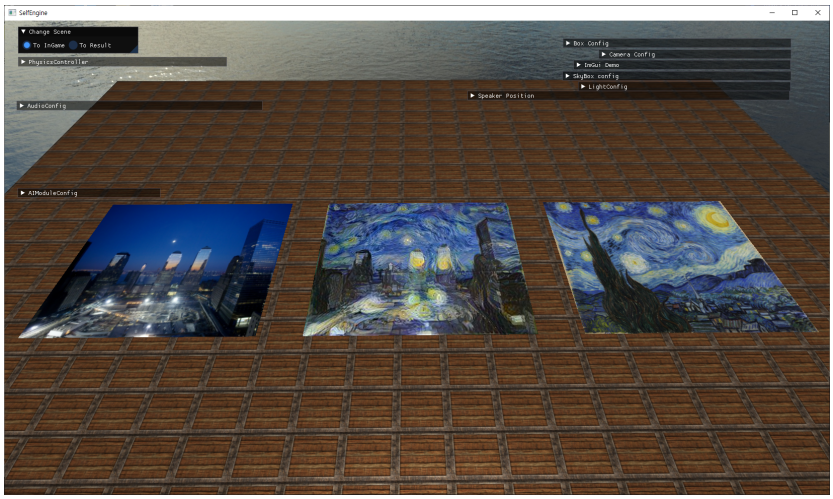
(**b**)

Figure 6. Screenshots from the bowling game. (**a**) Before a collision, (**b**) after the collision.



(**a**)

Figure 7. *Cont.*



(**b**)

Figure 7. Screenshots from a neural style transfer. (**a**) Before the style transfer, (**b**) after the transfer completed.

The used deep learning model for the neural style transfer is the sufficiently trained VGG19 model. This model consists of blocks, and the blocks consist of convolution layers. The layer used for a content layer is the convolution4 layer in the block 4, and the layers used for style layers are convolution1 layers in the blocks 1, 2, 3, 4 and 5. Figure 8a,b are the example content and style images. Figure 9 is the structure of the VGG19 (Visual Geomtry Group, Number 19) model.

(**a**) (**b**)

Figure 8. A set of images for the style transfer. (**a**) A content image. (**b**) A style image.

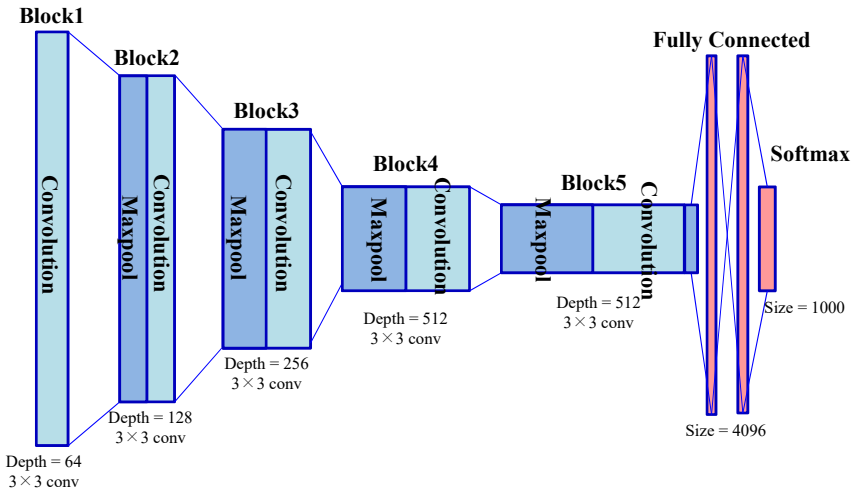


Figure 9. The structure of VGG19 (Visual Geomtry Group, Number 19).

4.1. DNN Module Comparisons

One distinguished point of our work is the DNN support. At least to our current knowledge, most of open-source game engines are lack of this new trend. In contrast, our game engine integrated the DNN module, and shows that we can use the pre-trained data, or the existing DNN models, for game applications.

In the case of commercial proprietary engines, they usually have plug-in supports, and recently they provide deep-learning related modules. In the case of Unity3D, they have the ML-agent (machine learning agent) module. However, this module is mainly focused on the reinforcement learning in the Unity3D framework. In the case of the Unreal Engine, they provide a TensorFlow plug-in for deep learning applications.

Table 2 represents the comparisons of the DNN supporting features, in our game engine, Unity3D, and the Unreal Engine. Both Unity3D and the Unreal Engine focused on the providing TensorFlow-based features to the game applications. Thus, as shown in Table 2, they provide Python language interface to the DNN modules.

Table 2. Comparisons on DNN (deep neural network) support.

	Our Game Engine (Neuro Module)	**Unity3D (ML-Agent)**	**Unreal Engine 4 (TensorFlow Plug-in)**
Platform	Windows PC	many Unity Build targets	PC, Android
Engine API languag	C++	C#/Java Script/Boo	C++
ML API language	C++	Python/TensorFlow	Python/TensorFlow
ML models	Keras	Unity Inference Engine/Custom Side Channel	Python scripts
Engine-ML communication	public queue	built-in script (RL situation)	SocketIO client plug-in with JSON
License	GPL-3.0	Apache 2.0	MIT/Apache 2.0

In contrast, our game engine uses the Keras-like C++ DNN module. Since the whole game engine and also the DNN module are implemented in C++ programming language, we have several benefits: In our case, the integration of the DNN module is intuitive, and the game engine and the DNN module can exchange any data with C++ data structures. In our implementation, we efficiently use a simple internal queue, which is public to all components of the game engine.

Notice that ML-agent in Unity3D provides Python interface, while the Unity3D engine itself is implemented in the C# and Java Script programming language. Their solution for the communication is introducing a new built-in script, with extra overhead for translating and mapping between difference programming languages. Similarly, the Unreal engine provides a set of Python scripts to process data exchanges between the C++-based Unreal engine and Python-interfaced TensorFlow plug-in.

For the learning process of neural networks, both of the commercial proprietary engines should use its own way of neural network training. Unity3D provides Jupyter notebook interface [34], while the Unreal engine has some custom scripts for this purpose. In our case, as shown in Section 4, the neural network training can be performed in a game application, with C++ API functions. Additionally, our DNN module also accepts pre-trained DNN data directly, while executing the game applications.

From the game application programmer's view, the use of another programming language and/or script languages can increase the complexity of the game development. In our case, the DNN module provides Keras-like C++-API, which can be easily accepted to the C++ programmers. It is one of the strong points in our implementation.

4.2. User Interfaces

Figure 10 shows the configuration UI windows of the audio and the physics module. By checking the RayCastApplyForce check box of the PhysicsController UI window, and setting x, y and z-axis of a force to apply, the mouse-clicking event on the screen calls Raycast function, which affects the defined force if the collided object has a rigid body. As for the audio UI, there are three buttons each for the play, pause and stop. Figure 11 shows the configuration windows of game objects. Changing the color and the position of the light is controlled by these UI windows. The console window in Figure 12 shows how much the neural style transfer training is processed.

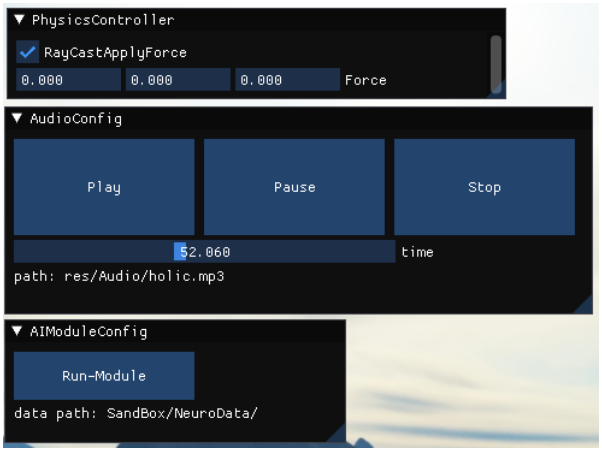


Figure 10. The UI (user interface) windows for physics, audio, and DNN modules, respectively.

Figure 11. A UI window for configuring a game object.

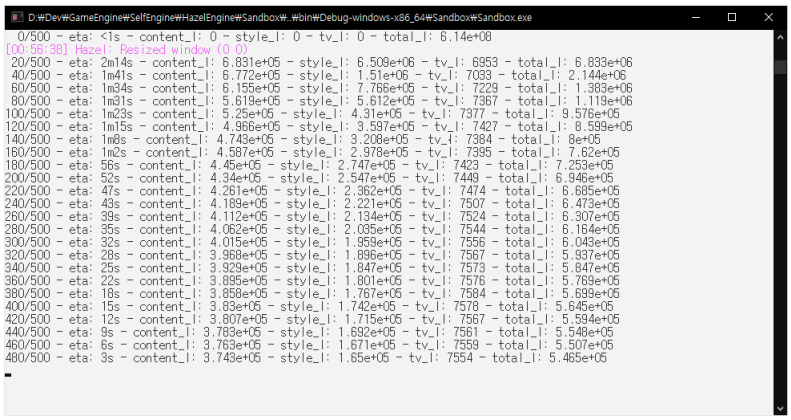


Figure 12. The console window output from the neural-network style transfer processing.

5. Technical Issues

This section addresses technical issues, raised during the development of the proposed game engine. Many problems are related to the merging process of different open source products.

5.1. Compatibility Issues with Data Structures

Since the proposed game engine integrates many open source products, there are some expressions conceptually the same but defined as different data types. For example, for the same three-dimensional vector type, the physics module uses rp3d::vector3 type and rending module uses glm::vec3 type. Conceptually, both types are the same, but cannot be used interchangeably without overloading the assignment operators. Another case is the texture image handling. The texture class implemented in the proposed game engine conflicted with the texture structure of the open asset import library. Empirically, internal types of similar purpose open source products often collide, and we need to provide type conversion solutions for these type compatibility issues.

5.2. Library Conflicts

Conflict occurs between libraries, if code generate options are different. A simple and often conflict occurs in the choice of debugging libraries. Some open source projects wish to use debugging mode libraries, such as the MDd (multi-threaded, dynamically linked, debug mode) library in VisualStudio, while others prefer the non-debugging multi-threaded, dynamically linked (MD) library. We need to integrate both cases into a single one executable file, and thus, we carefully investigated each open source projects, and finely tuned to use non-debugging ones, to avoid any library conflicts.

5.3. Name Collisions

During integration of many open source projects, we often met the name collisions, especially for file names and class names. As an actual example, ambiguity occurred when integrating the physics module and the DNN module, since both module has their own MemoryManager.cpp files. Of course, we fixed these collisions through simply renaming the file names. In the case of class name collisions, we simply use C++ namespaces.

5.4. Path Configurations

When an open source project has been carefully configured with relative paths to the source files, header files, and library files, it is easy to integrate that open source project to other systems. In contrast, some absolute paths can cause extra operations. Some open source projects use CMake [35] for compiler configurations. We found that CMake can use different path configurations, depending on the CMake versions. When an open source project depends on a specific version of CMake, we should resolve those dependencies, or copy the CMake Binary files to the exactly same paths.

5.5. Poor Documentation

Rarely used or poorly managed open source projects may have poor documentation. In this case, programmers should read source codes directly, and infer how to use it. However, it takes a lot of time to resolve, or even have to give up to use it, if the developer did not specify proper precautions. For example, MP3 files have various sample rates, while the open source library used by the audio module accepts 24,000 Hz, 44,100 Hz, and 48,000 Hz sampling rates only. Unaware of these precautions might cause difficulties to guess what values accepted.

5.6. Version Compatibility

Some open source projects may depend on specific versions of programming languages. In fact, the proposed game engine initially used the string-view features in the C++17. However, we found that the DNN module uses some features in the earlier versions of C++. To resolve this problem, we finally used C++14, removing some C++17 features.

6. Conclusions

This paper shows the details of developing a light-weight game engine with open source products. The overall structure and components of the proposed game engine were explained. Additionally, we focused on the technical issues, while integrating many open source products into a single project result. Using open source products require various considerations, as shown in Section 5.

As a case study, a simple demonstration game program shows the possibility of the DNN methods, to generate or to modify game objects in real-time. One remarkable benefit of our implementation is that we can use the same C++ programming language for both of the game application development and also the DNN methods, while other game engines need extra programming language interfaces and/or script modules. It gives us more efficient and flexible way of developing game applications with DNN supports.

In the near future, we will focus on the improvement of the game engine structures, to enhance the game engine to fully utilize the DNN module in various ways with clearly defined internal structures. Simultaneously, we will utilize this game engine to integrate deep learning tasks with games to develop new features. As an example, it can include creating or mutating objects in a game using a generative adversarial network (GAN) [36], or creating and playing background music to match the internal background of a game, and so on.

Author Contributions: Conceptualization, H.P. and N.B.; implementation, and writing—original draft preparation, H.P.; writing—review and editing, supervision, project administration, and funding acquisition, N.B. All authors have read and agreed to the published version of the manuscript.

Funding: This work has supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (Grand No. NRF-2019R1I1A3A01061310). This research was also supported by the BK21 Plus project (SW Human Resource Development Program for Supporting Smart Life) funded by the Ministry of Education, School of Computer Science and Engineering, Kyungpook National University, Korea (21A20131600005).

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

References

1. Méndez-Suárez, M.; García-Fernández, F.; Gallardo, F. Artificial Intelligence Modelling Framework for Financial Automated Advising in the Copper Market. *J. Open Innov. Technol. Mark. Complex.* **2019**, *5*, 81.
2. Lee, J.; Suh, T.; Roy, D.; Baucus, M. Emerging Technology and Business Model Innovation: The Case of Artificial Intelligence. *J. Open Innov. Technol. Mark. Complex.* **2019**, *5*, 44. [CrossRef]

3. Lee, M.; Yun, J.J.; Pyka, A.; Won, D.; Kodama, F.; Schiuma, G.; Park, H.; Jeon, J.; Park, K.; Jung, K.; et al. How to Respond to the Fourth Industrial Revolution, or the Second Information Technology Revolution? Dynamic New Combinations between Technology, Market, and Society through Open Innovation. *J. Open Innov. Technol. Mark. Complex.* **2018**, *4*, 21. [CrossRef]
4. Unity Technologies Blog. Available online: http://blogs.unity3d.com/category/machine-learning/ (accessed on 11 June 2020).
5. Unit Machine Learning Agents Toolkit. Available online: http://github.com/Unity-Technologies/ml-agents (accessed on 11 June 2020).
6. TensorFlow Plugin for Unreal Engine 4. Available online: http://github.com/getnamo/tensorflow-ue4 (accessed on 11 June 2020).
7. Romhányi, A.; Szénási, S. OpenGL-based Modular Lightweight 3D Framework with Physics Capabilities. In Proceedings of the 2019 IEEE 17th World Symposium on Applied Machine Intelligence and Informatics (SAMI), Herlany, Slovakia, 24–26 January 2019; pp. 239–244.
8. Navarro, A.; Pradilla, J.V.; Rios, O. Open source 3D game engines for serious games modeling. In *Modeling and Simulation in Engineering*; Hindawi Publishing Corporation: London, UK,2012; pp. 143–158.
9. Gregory, J. *Game Engine Architecture*, 3rd ed.; CRC Press: Boca Raton, FL, USA, 2018.
10. Gatys, L.; Ecker, A.; Bethge, M. A Neural Algorithm of Artistic Style. *J. Vis.* **2016**, *16*, 326. [CrossRef]
11. Wikipedia: Software Engine. Available online: http://en.wikipedia.org/wiki/Software_engine (accessed on 11 June 2020).
12. What is a Game Engine? Available online: http://www.gamecareerguide.com/features/529/what_is_a_game_.php (accessed on 11 June 2020).
13. Pargman, D.; Jakobsson, P. Five Perspectives on Computer Game History. *Interactions* **2007**, *14*, 26–29. [CrossRef]
14. Unreal Engine. Available online: http://www.unrealengine.com/ (accessed on 11 June 2020).
15. Unity Realtime Platform. Available online: http://www.unity.com/ (accessed on 11 June 2020).
16. CryEngine. Available online: http://www.cryengine.com/ (accessed on 11 June 2020).
17. Amazon Lumberyard. Available online: http://aws.amazon.com/lumberyard/ (accessed on 11 June 2020).
18. Cocos2D. Available online: http://www.cocos.com/ (accessed on 11 June 2020).
19. Godot Engine. Available online: http://www.godotengine.org/ (accessed on 11 June 2020).
20. Gdevelop. Available online: http://www.gdevelop-app.com/ (accessed on 11 June 2020).
21. Bertens, P.; Guitart, A.; Chen, P.P.; Perianez, A. A Machine-Learning Item Recommendation System for Video Games. In Proceedings of the 2018 IEEE Conference on Computational Intelligence and Games (CIG), Maastricht, The Netherlands, 14–17 August 2018; pp. 1–4.
22. Abdollahi, M.; Dadkhah, C. Intelligent Android Game using Reinforcement Learning to Change the Enemy's Behavior. In Proceedings of the 2018 2nd National and 1st International Digital Games Research Conference: Trends, Technologies, and Applications (DGRC), Tehran, Iran, 29–30 November 2018; pp. 172–179.
23. Tilson, A.; Gelowitz, C.M. Towards Generating Image Assets Through Deep Learning for Game Development. In Proceedings of the 2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE), Edmonton, AB, Canada, 5–8 May 2019; pp. 1–4.
24. OpenGL Web Site. Available online: http://www.opengl.org/ (accessed on 11 June 2020).
25. ImGui: Bloat-Free Immediate Mode Graphical User Interface for C++ with Minimal Dependencies. Available online: http://github.com/ocornut/imgui (accessed on 11 June 2020).
26. Segal, M.; Akeley, K. *The OpenGL Graphics System: A Specification*; Khronos Group: Beaverton, OR, USA, 2019.
27. The Open-Asset-Importer-Lib. Available online: http://www.assimp.org/ (accessed on 11 June 2020).
28. OpenGL Mathematics. Available online: http://glm.g-truc.net/0.9.9/index.html (accessed on 11 June 2020).
29. ReactPhysics3D. Available online: http://www.reactphysics3d.com/ (accessed on 11 June 2020).
30. Portaudio—An Open-Source Cross-Platform Audio API. Available online: http://www.portaudio.com/ (accessed on 11 June 2020).
31. The Cross-Platform Audio Decoder API. Available online: http://github.com/asantoni/libaudiodecoder/ (accessed on 11 June 2020).
32. Steam Audio. Available online: http://valvesoftware.github.io/steam-audio/ (accessed on 11 June 2020).
33. Neuro library. Available online: http://github.com/Cr33zz/Neuro (accessed on 11 June 2020).
34. Jupyter Notebook. Available online: http://www.jupyter.org/ (accessed on 11 June 2020).

35. CMake. Available online: http://cmake.org/ (accessed on 11 June 2020).
36. Goodfellow, I.J. NIPS 2016 Tutorial: Generative Adversarial Networks. *arXiv* **2016**, arXiv:1701.00160.