Korean Society For Computer Game, Vol. 30, No. 1, March 2017, pp.25-34
ISSN 1976-6513 Copyright ⓒ 2017 KSCG http://dx.doi.org/10.21493/kscg.

코드 재사용을 위하여 컴포넌트 방식을 사용하는 DirectX 11 기반 3D 게임 연구

김영식

A Study of DirectX 11 based 3D Game using Component Method for Code Reuse

Youngsik Kim*

Department of Game and Multimedia Engineering, Korea Polytechnic University,
237 Sangidaehak-ro, Siheung-City, Gyeonggi-Do 429-793, Korea

(received Fabruary 10, 2017; revised March 22, 2017 ; accepted March 29, 2017)

ABSTRACT

In commercial game engines, a method of constructing objects by combining components is often used. Components can be assembled and used at any time as needed. In this paper, the 3D game is developed by applying the component method, which is the object creation method widely used in the commercial game engine, in the DirectX 11 environment. There are four kinds of components used in the in-house 3D game. These are four components such as transform, mesh, script, and collider. The component generation method used in this paper shows very good operation efficiency in terms of reusability. This paper also applied tangent space normal mapping to static and dynamic objects and analyzed performance at various screen resolutions. Performance analysis showed that the average rendering speed was 64.6% higher on the low resolution screen than on the high resolution screen. And the rendering speed of model G with normal mapping to all objects among the 8 simulation models is improved by 19.2% compared with model B without normal mapping. The presence or absence of normal mapping has the greatest effect on rendering speed.

Key words: DirectX 11, 3D Game, Code Reuse, Components, Tangent Space Normal Mapping, FPS(Frame per Second)

1. Introduction

Most game development in the past had to develop all the systems necessary for the game, but in recent years, the time required for development has been reduced to less than half, because only the specific game system parts required for the development of the game engine have to be developed [1] [2]] [3] [4] [5] [6].

* Correspondence to: Youngsik Kim, Tel.: +82-31-8041-0560 E-mail: kys@kpu.ac.kr

The core functional blocks of commercial game engines such as, Unity 5 [4], Unreal Engine 4 [5], and Amazon Lumberyard [6], as shown in Fig. 1, consists of a rendering engine, a physical engine, and a scene manager. [4] [5] [6].

In large-scale online 3D game development, interaction between objects is a very important factor and affects the performance of clients. The method of composing objects by combining components is often compared to the LEGO block. As it can be assembled and used at any time as needed, it is very helpful to improve code reuse and game development productivity.

Table 1 compares Unity 5 [4], Unreal Engine 4 [5], and Amazon Lumberyard [6], typical component-based commercial games. The workflow of the Unity 5 [4] game engine is based on Prefab. Create a GameObject with a component and make it with Prefab. Instances of Prefab can be placed in the game space or instantiated at runtime.

All modifications will be reflected in the created Prefab instance, but can be set individually for each instance. The Unity 5 [4] game engine constructs a hierarchy of GameObjects and creates objects with transforms (Fig. 1). We add a C # script to the GameObject for the script component, and create a class that inherits from MonoBehaviour to define the functionality of that component.

The development workflow for Unreal Engine 4 [5] is based on Blueprint. Blueprint is a visual scripting system based on the concept of creating gameplay elements using a node-based interface. The basic form is added to the game with visual scripting, which makes it possible to create complex gameplay elements by connecting nodes, events, functions, and variables with lines. Commonly used Blueprint types are Level Blueprint and Blueprint classes. Components cannot exist independently and must be added to an actor. For example, a Spotlight component can cause an actor to light up like a spotlight, and a Rotating Movement component to cause an actor to rotate. When you add a component to an actor, you put together the pieces that make up the entire actor. For example, a car, a steering wheel, a car body, a light, etc., are regarded as components , whereas a car itself is regarded as an actor.

The Flow Graph of the Amazon Lumberyard [6] game engine consists of nodes and links as a visual scripting system that can implement complex game logic without having to organize code. A node may represent a Level (Entity level) or an Action (component Node) that performs specific operations on a target entity. A link is used to connect nodes and is represented by lines connecting input / output between nodes. The node library provides everything needed to fully control objects and AI agents within the level. The script of the Lumberyard engine consists of four categories.

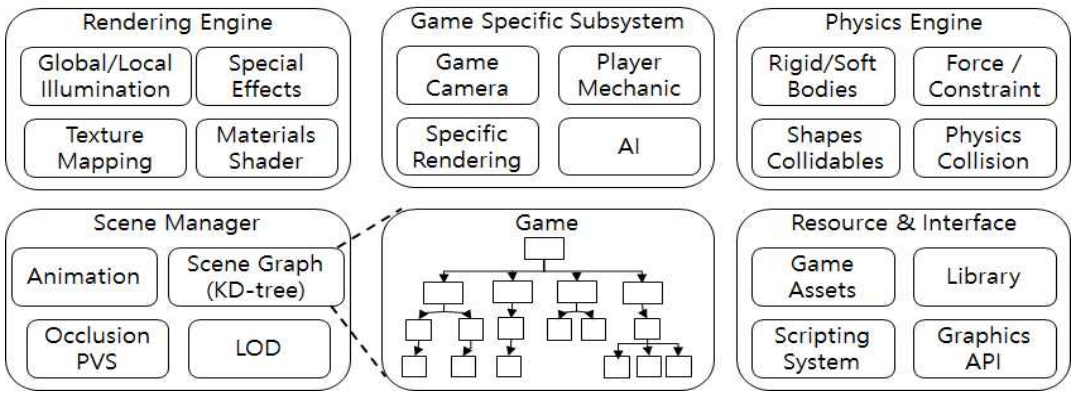

Fig. 1. Game Engine Architecture

Table 1. Comparison of Commercial Game Engines

Category	Unity 5 [4]	Unreal Engine 4 [5]	Lumberyard [6]
GamePlay Type	Component	Component	Component
	GameObject	Actor, Pawn	Object
	Prefab	BluePrint	Flow Graph
Editor UI	Hierarchy	World Outliner	Entity Outliner
	Inspector	Details Panel	Entity Inspector
	Project Browser	Content Browser	File Browser
	Scene View	Viewport	Viewport
Meshes	Mesh	Static Mesh	Static Mesh
	Skinned Mesh	Skeletal Mesh	Skeletal
Materials	Shader	Material	Shader
	Material	Material Instance	Material
Effects	Particle Effect	Effect, Particle, Cascade	Particles, ParticleImposter
Game UI	UI	UMG	UI
Animation	Animation	Skeletal Animation System	Skeletal Animation
	Mecanim	Persona, Animation BluePrint	Geppetto
Programming	C#	C++	C++
Physics	Raycast	Line Trace, Shape Trace	Physics Proxies
	Rigid Body	Collision, Physics	Basic Entity, Rigid Body

- Level - Contains a unique script file for the current level. (configured in Entities, Components, Modules)
- Global
- Prefab - Allows direct connection between instances. Generally, when you perform an operation, you create an event inside the Prefab, name it, and then reference that Prefab instance.
- External Files

In the game, bump mapping is an efficient way to use low-resolution meshes and store high-resolution mesh features on bumpy surfaces in textures for runtime use [7] [8] [9] [10] [11]. Three methods for handling bump mapping using a height map are (1) normal mapping, (2) parallax mapping, and (3) displacement mapping. (1) Normal mapping is the oldest and most used in the game. First, in the preprocessing step, the height map is used to calculate the perturbed normal of the bumpy surface and store it in a special texture called normal map.

(2) Parallax mapping [9] performs a simple ray tracing algorithm on the height map at runtime. Unlike normal mapping, it is possible to express that the bump is actually hidden. We fire one ray per pixel, calculate the point at which this ray hits the height map, and determine the color of that point. (3) Displacement mapping [9] actually tessellates the macro structure and then uses the height map to actually move the vertices. Since it supports tessellation in shader model 5, more attractive algorithm displacement mapping is implemented with tessellator and domain shader.

In this paper, we demonstrate the effectiveness of applying component methods for code reuse in commercial game engines to 3D game development DirectX 11. In addition, we adopt low-resolution mesh, and use tangent space normal mapping which minimizes the computation by using high-resolution mesh features in texture, and storing them in runtime. It is designed to be applied to both static and dynamic objects in self-produced 3D games.

This paper consists following composition. First in Section 2, we describe the overall structure of 3D games, component-based design, and tangent

space normal mapping rendering techniques. In Section 3, we compare and analyze various rendering effects based on normal mapping. Finally, the conclusion of this paper is described in the last section

2. DirectX 11-based, built-in 3D game

2.1 Self-built 3D game structure

In this paper, we develop a 3D game based on DirectX 11 using component method for code reuse and apply tangent space normal mapping rendering effects to static and dynamic objects and analyze performance at various screen resolutions.

The DirectX 11-based 3D game used in the experiment is a self-produced online action RPG game. It uses a third person's back view and clears the monsters that are in the map with the party member, and finally defeat the boss.

Fig. 1 describes the operation flow of self-produced 3D game. When you first run the client, you first connect to the server using the Windows Socket Model IOCP (Input Output Completion Port) to join the stage. IOCP is a way to limit the creation of threads to work, reduce thread context switching costs and thread creation costs, and manage threads, to handle input and output (I / O) [12]. Each client is continuously synchronized using an event method while the game is running through the server. In other words, when an event requiring synchronization is generated, information necessary for synchronization is transmitted in a socket so that information of other clients is simultaneously updated. The update period may be different depending on the type of information. For example, the movement and rotation of a character is very fast because the cycle of changing values is very fast, but however the change of the state value and the application of the damage value are relatively short in the movement of synchronization data, because the event generation period is relatively long.

Table 2 shows the number of vertices and triangles of dynamic objects used in the game. Table 3 describes the static objects used in the game.

Table 2. Dynamic Objects

	Player(Female)	Player(Male)
No. of Vertices	4939	4942
No. of Triangles	5941	4954

Table 3. Static Objects

	Ruin1	Ruin2	Ruin3	Ruin4
No. of Vertices	7422	6024	3660	2942
No. of Triangles	5560	4641	2633	2127

As shown in Fig. 3, the game progresses from the starting point towards the village in the direction of the enemy. Once you reach the entrance to the village, a young man is met by the monsters, and if you can defeat the enemy, you can proceed to the next level. The village in the desert is attacked by monsters and you are asked to kill monsters. If you follow the road leading to the village, you will meet the boss, Dragon King. And if one can kill the boss, then the game is cleared.

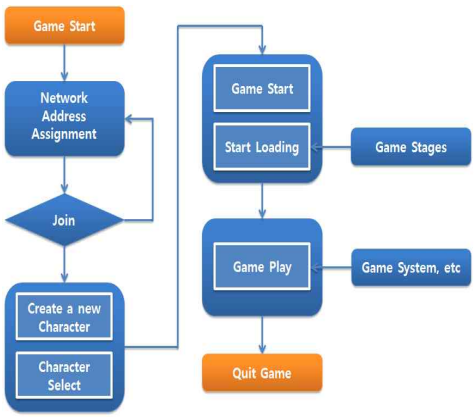

Fig. 2. Control Flow

(a) Dynamic and Static Objects

(b) Boss Monster

Fig. 3. Game Screen Shots

2.2 Component based 3D game

In this paper, there are four kinds of components used for code reuse: Transform, Mesh, Script, and Collider (Fig. 4). The implementation of object creation using components uses a template, and a singleton pattern.

Every object that exists on it has one transform component with position and rotation values. The other components can be added to the object without any limitation. This is because the STL (Standard Template Library) <map> container is implemented by adding and managing component names as key values.

Transform components have object size, position, and rotation information. There is a world matrix that integrates all the transform information, and the transform matrix has local matrix information for the objects connected by the inheritance structure. Transform components cannot be added randomly, since only one transform property can be used per object.

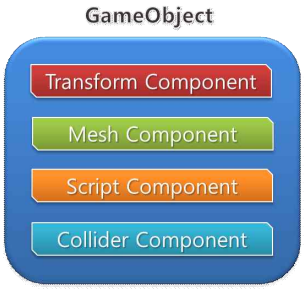

Fig. 4. Component Structure for Game Objects

Mesh components have information about the shape of an object that is visible to our eyes. Almost all objects except the camera, the collision object, and the dummy object have the shape. And that form can be as still as a building or weapon, or can have movement as a character. It is necessary to set the constant buffer to be used differently, and the static mesh and the skinned mesh are separately managed so that the application of the shader can be separately performed later. Since we can use the same mesh several times, we create a manager class that can be stored separately using a STL <map> container, like a component, and implemented it so that it can be used afterwards if necessary. In this project, one model file has all the state animation information, and the frame range of the animation is stored and reproduced according to it. When the animation corresponding to the current state is finished, the state transition is selected depending on whether the animation is to be reproduced again or another new animation is to be reproduced which is associated with the script component to be described below.

The script component has script information to change the transform information or state information of the object. For example, in the case of a player's character, information about various state changes, such as moving or rotating by pressing a directional key, hitting an enemy when

hit by an enemy, and HP reduction, are pre-written in the script and added to the object as a script component. Rather than writing one script for each of the many enemy movements, if you add these pre-written scripts as components, the applied objects will all take the same action. Of course, with random values, you can also use the same script to get a different state transformation. In the case of a camera, a script for the boss appearance scene can be created in advance, and the script can be executed when a specific condition is reached.

The collider component has information for collision checking. There are many conflict handling methods such as sphere collision and box collision. In Saviors project, the most simple collision detection method is sphere collision. Therefore, the collider component has information about the radius of the collision sphere, performs collision checking using it, and changes the state of the collision according to the script component.

Objects created by combining these components can also be pasted using the inheritance structure. Although not mentioned previously, the component also has its own top-level class and inherits the class to define the component in detail in the subclass (Fig. 5).

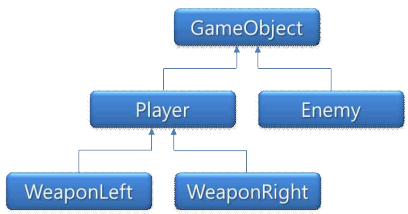

Fig. 5. Inheritance Structure for Game Objects

The game object class, which is higher than the components, has pointer variables of another game object here. You can use this to set the parent-child relationship of the object and use it in the Saviors project to attach to the weapon of the player's character. The object weapon has its own local matrix information, and the world matrix is determined by multiplying the world matrix of the inorganic bone of the character and the parent object, by the local matrix of the weapon. The weapon moves along with the hand position of the character making it appear as the character is holding the weapon by hand.

I'll finish the explanation of the composition of the object and talk about the interaction of the object. Interaction between representative objects in RPG is an action between player character and enemy monster. The process of detecting the player by calculating the distance, finding out which side it is, rotating, chasing after entering the battle, etc., needs transform information of these objects.

Now, we finish the description of the composition of the object is here, and talk about the object interaction.

Fig. 6 shows the player and the enemy from above. The black arrow indicates the direction vector of each object, and the blue dotted line indicates the direction corresponding to the difference between the enemy vector and the player vector. The yellow circle surrounding the enemy is the enemy player's detection range. That is, when the player enters the yellow circle, the enemy enters the aggro state. When entering the aggro state, first take the aggro motion once and then turn to the running state while rotating in the direction of the player. Assuming that in the situation shown in the picture, the player goes straight in the current direction and enters the detection range of the enemy, it must be rotated clockwise to reach the player in the optimal path. To make this judgment, external cross products are used in enemy scripts. The position vector of the player is subtracted from the enemy position vector, and the vector value is external to the enemy direction vector. If the y component of the vector thus calculated is a positive number, it rotates in the counter clockwise direction and if it is negative, it rotates in the clockwise direction, causing the enemy to turn to the optimal path.

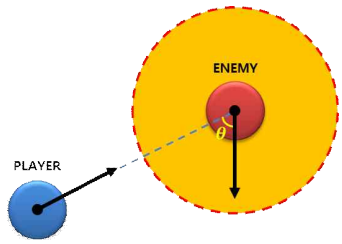

Fig. 6. Detect Enemy

2.3 Tangent space normal mapping

In the games, the normal mapping technique is an efficient way to use low-resolution meshes and store high-resolution mesh featuring on bumpy surfaces in textures for runtime use.

In this paper, we adopt a low - resolution mesh, and use tangent space normal mapping which minimizes the computation by storing features of high - resolution mesh in texture and using it at runtime. It is designed so that it can be applied to both static and dynamic objects in self-produced 3D game (Fig. 7).

Like the mesh, the shader also has a manager class to store and manage. Once created, shader information can be reused whenever there is a need to create another object (Fig. 8).

The light vector of the bump mapping contains a normal vector, which determines the direction in which the light is reflected. That is, by using the normal map generated by using the texture, it is possible to shake the normal vector of the object surface to look more stereoscopically. It allows them to look more three-dimensional. To apply a normal mapping to a dynamic object, a tangent space basis {T, B, N} is defined. The vertex normal N is defined vertically at the modelling stage, and the tangent T and the binormal B are calculated as following equations [Eq. 1,2,3].

$$N(u,v) = \frac{dP(u,v)}{du} \times \frac{dP(u,v)}{dv} \quad \text{[Eq.1]}$$

$$T \propto \frac{du}{dP} = \left(\frac{du}{dx}, \frac{du}{dy}, \frac{du}{dz} \right) \quad \text{[Eq.2]}$$

$$B \propto \frac{dv}{dP} = \left(\frac{dv}{dx}, \frac{dv}{dy}, \frac{dv}{dz} \right) \quad \text{[Eq.3]}$$

(a) With Normal Mapping

(b) Without Normal Mapping

Fig. 7. Game Screen Shots with/without Normal Mapping

```
f4BumpMap = (bumpMap * 2.0f) - 1.0f;
f3BumpNormal = (f4BumpMap.x * tangent)
             + (f4BumpMap.y * binormal)
             + (f4BumpMap.z * normal);
f3BumpNormal = normalize(f3BumpNormal);
```

Fig. 8. Shader Codes for Normal Mapping

Table 4. Experimental Models

Experimental Model	A	B	C	D	E	F	G	H
Number of Dynamic Objects	16	32	16	32	16	32	16	32
Normal Mapping on Dynamic Objects	Yes				No			
Number of Static Objects	20							
Normal Mapping on Static Objects	Yes		No		Yes		No	

Table 5. Frame Per Seconds (FPS) of Experimental Models to Various Screen Resolutions

Screen Resolutions \ Models	A	B	C	D	E	F	G	H
640x360	635.7	585.7	698.0	629.0	641.1	594.4	700.3	643.6
1024x576	490.5	448.0	536.4	488.3	503.0	467.9	540.5	507.5
1366x768	324.7	300.2	353.5	328.7	334.2	308.9	361.2	335.0
1600x900	268.5	249.3	287.8	265.4	271.1	254.7	293.9	272.7
1920x1080	226.4	210.0	240.1	223.8	229.0	215.3	241.1	229.7

3. 3D Game Experiment and Verification

Performance analysis was performed by measuring the rendering speed (FPS: frame per second). And for this, the performance of the computer used in the experiment has the processor: Inter (R) Core i5 4210M @ 2.60GHz 2.60GHz, memory: 4.00GB, 64-bit operating system, graphics card: Nvidia Geforce 940m. Also, for the rendering performance analysis, the screen resolution of the device was measured while changing to 640x360, 1024x576, 1366x768, 1600x900, 1920x1080, and so on. Table 4 defines eight simulation models with varying numbers of static and dynamic objects and with or without tangent space normal mapping.

As can be seen from the comparison of the rendering speeds in Table 5, the lower the screen resolution, the higher the rendering speed was achieved in all eight models.

The average rendering speed for 640x360 resolution was 64.6% higher than 1920x1080 resolution. Standard

deviations were 41.7 and 10.8 at 640x360 and 1920x1080 resolutions respectively, with a standard deviation of being higher up to 74.1% at lower resolutions.

The average rendering speed according to the various resolutions of eight simulation models was higher than the other models with the model G average speed of 427.4 FPS. The reason that the rendering speed of Model G is higher than other models is that normal mapping is not applied to both static and dynamic objects, and the number of dynamic objects is also small. The average rendering speed of Model G is 19.2% higher than that of B model with normal mapping for both static and dynamic objects.

4. Conclusion

In this paper, we develop a 3D game based on DirectX 11 using component method for code reuse and apply tangent space normal mapping rendering effects to static and dynamic objects and analyse performance at various screen resolutions.

The component generation technique shows very good work efficiency in terms of reusability. All objects are created and managed using components and inheritance structures, which makes it easy to create a large number of objects, and it is very convenient to implement interactions between objects, as they are very clear and intuitive in their code. At the same time, the object-oriented nature of the C ++ language combines to create synergistic effects and facilitate collaboration. Normal mapping performance analysis showed that the average rendering speed was 64.6% higher on the low resolution screen than on the high resolution screen. And the modelling speed of model G with normal mapping to all objects among the 8 simulation models is improved by 19.2% compared with model B without normal mapping. The presence or absence of normal mapping has the greatest effect on rendering speed.

The implication is that the number of static and dynamic objects and the rendering speed according to the screen resolution should be considered whether or not the tangent space normal mapping is applied in the DirectX 11 based 3D game development process. In the future, research can be expanded by applying parallax mapping and displacement mapping.

4. 결론

본 논문에서는 코드 재사용을 위하여 컴포넌트 방법을 사용하는 DirectX 11 기반의 3D 게임 개발하고 탄젠트 공간 노멀 매핑 렌더링 효과들을 정적 및 동적 오브젝트에 적용하고 다양한 스크린 해상도에서 성능 분석 하였다.

컴포넌트 생성 기법은 재사용성 측면에서 볼 때 매우 우수한 작업 효율을 보인다. 모든 객체들은 컴포넌트 및 상속 구조를 이용하여 생성 및 관리되며, 이 때문에 많은 수의 객체 생성도 용이하며, 각 객체간의 상호작용을 구현할 때도 코드 상으로도 매우 명확하고 직관적이었기 때문에 상당히 편리하다. 이와 동시에 C++ 언어의 객체지향적인 특성이 결합되어 시너지 효과를 내며, 공동 작업 또한 수월하게 진행될 수 있다.

노멀 매핑 성능 분석 결과 평균 렌더링 속도가 고해상도 스크린에 비하여 저해상도 스크린에서 64.6% 높았다. 그리고 8가지 시뮬레이션 모델 중에서 모든 오브젝트에 노멀 매핑을 적용한 모델 G의 렌더링 속도가 노멀 매핑을 적용하지 않은 모델 B에 비하여 19.2% 향상되었다. 노멀 매핑 적용 유무가 렌더링 속도에 가장 큰 영향을 미쳤다.

시사하는 점은 DirectX 11 기반 3D 게임 개발 과정에서 탄젠트 공간 노멀 매핑의 적용 유무를 정적 및 동적 오브젝트의 개수와 스크린 해상도에 따른 렌더링 속도를 고려해야 한다. 향후에는 패럴랙스 매핑과 변위 매핑을 적용하여 연구를 확대할 수 있다.

ACKNOWLEDGEMENT

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No. 2016-0-00204, Development of mobile GPU hardware for photo-realistic real time virtual reality).

참고문헌

1. Dongryul Lee and Youngsik Kim, "Comparison of 3D Game Visual Effect Techniques based on Deferred Rendering using Multiple Render Targets", *Journal of Korean Society for Computer Game*, Vol.28, No.4, pp. 1-10, 2015.
2. C. J. Lim, Won Dae Han, Jeong Yun Guen, “Educational Game Making-Tool Development using Unity3D Engine: Birth of Game”, *Journal of Korea Game Society*, Vol. 14, No. 1, pp.29 ~38, 2014.
3. Kun Sung Lee and Youngsik Kim, "Implementation of A Stereoscopic 3D Game To Reduce Visual Artifacts Using Oculus Rift", *Journal of Korean Society for Computer Game*, Vol.27, No.4, pp. 193-201, 2014.
4. Unity3D Game Engine, https://unity3d.com/kr/
5. Unreal Engine 4, http://unrealengine.com/
6. Amazon Lumberyard Game Engine, https://aws.amazon.com/ko/lumberyard/
7. Mark J. Kilgrd, “A Practical and Robust Bump-mapping Technique for Today’s GPUs,”, *Game Developers Conference 2000*.
8. Mark Peercy, John Airey, and Brian Cabral, “Efficient Bump Mapping Hardware”, *Computer Graphics (Proc. Siggraph ‘97)*: pp. 303~306.
9. JungHyun Han, *3D Graphics for Game Programming*, CRC Press, 2011.
10. Wofgang Engel, *Shader X5 Advanced Rendering Techniques, 2.6 Normal Mapping without Precomputed Tangent*s, Charles River Media, 2007.

11. James Blinn, “Simulation of Wrinkled Surfaces,’’ *Computer Graphics (Proc. Siggraph ’78),* August 1978, pp. 286-292,. Also in *Tutorial: Computer Graphics: Image Synthesis*, pp. 307-313.
12. Jeffrey Richter and Christophe Nasarre, *Windows via C/C++*, Vol. 4, Microsoft Press, 2008.

Youngsik Kim

Youngsik Kim received the B.S., M.S., and Ph.D degree in Dept. Computer Science from the Yonsei University, Korea, in 1993, 1995, and 1999 respectively. He had worked for System LSI, Samsung Electronics Co. Ltd from Aug. 1999 to Feb. 2005 as a senior engineer. Since March 2005 he has been working for Dept. of Game & Multimedia Engineering in Korea Polytechnic University. His research interests are in 3D Graphics and Multimedia Architectures, Game Programming, and SOC designs.