GF Engine

A Versatile Platform for Game Design and Development

Hock Soon Seah, Henry Johan, Chee Kwang Quah, Nicholas Mario Wardhana, Tze Yuen Lim and Darren Wee Sze Ong

Abstract The GF Engine is conceived to be a versatile platform for game design and development that is based on best-of-breed game components and technologies to facilitate related R&D activities as well as application and content development. The engine aims to provide a framework for the research community, from researchers, technologists, application developers, and content creators alike, to collaborate, share, and experiment with innovative ideas and creative endeavors. At the research end of the spectrum, researchers conducting R&D on specific components could utilize the GF Engine as an integrating environment for proof-of-concept validation. At the application end of the spectrum, content developers could turn to the GF Engine to create their applications and contents efficiently.

Keywords Game engine · Best-of-breed components · Future-proof · Abstraction and encapsulation · Rendering · Physics · Scripting

1 Introduction

The GF Engine, abbreviated here as GF, is a versatile gaming platform where the research community may use as a framework to experiment and innovate their research. At the same time, developers, game designers, and gamers can create or modify a game in an intuitive manner. GF does not reinvent the wheels but instead takes advantage of the state-of-the-art open source software packages. However, it

H. S. Seah (✉) · H. Johan · C. K. Quah · N. M. Wardhana · T. Y. Lim
School of Computer Engineering, Nanyang Technological University, Singapore, Singapore
e-mail: ashsseah@ntu.edu.sg

D. W. S. Ong
DSO National Laboratories, Singapore, Singapore

Y. Cai and S. L. Goei (eds.), *Simulations, Serious Games and Their Applications*,
Gaming Media and Social Effects, DOI: 10.1007/978-981-4560-32-0_10,
© Springer Science+Business Media Singapore 2014

will not be totally dependent on these packages. It builds wrappers around them and grows with their future additional features. Its modular architecture allows replacement of individual or all components it is built upon.

Up to now, making game requires intensive collaboration between game designers and game developers. By decreasing the involvement of developers in the pipeline, GF plays the role of the developers. This is a big challenge, especially when GF wants to achieve the same level of runtime performance provided by other specialized game engines. To support rapid game development, GF is divided into two layers of abstraction, the *core engine* for general purpose game programming and the *template-specific API* for rapid construction of a well-defined game genre.

The rest of this chapter is organized as follows. Section 2 discusses the objectives of GF, what is it conceived to be. This is followed by the design approaches behind GF in Sect. 3. In Sect. 4, the architecture of GF is discussed. Details of some of the components in the architecture are explained in Sect. 5. An application and coding example is given in Sect. 6, and we conclude in Sect. 7.

2 GF Engine Objectives

2.1 Evolving Platform

Studying the architecture of present game engines and simulation software, we can summarize that game engines consist of the following components which are widely available as standalone packages: graphics renderer, physics simulator, artificial intelligence module, networking, and asset database.

GF, as a game development platform, also contains the above-mentioned components. GF integrates and manages the interaction among these components in a way such that the game development process can be done as easily as possible. In addition, GF, aiming to stay future-proof, is designed based on a modular architecture. Various components in GF can be replaced, allowing GF to evolve and stay up to date.

Using abstraction and encapsulation techniques, class wrappers are implemented around each component to isolate them from the rest. These modules are integrated in a well-defined structure, which is applicable to many components currently available in both commercial and open source packages. The features of these components are manifested through the core of GF, i.e., the Sandbox Integrated Development Environment (IDE).

2.2 Platform for Everyone

Game programming is a complex combinatorial problem. Creating games, especially networked games, is very challenging. A typical decent game studio consists of many skillful programmers, artists, and game designers working together in a sophisticated work flow. Even experienced programmers or scientists from different fields find it difficult to create game or simulation for visualizing their experiment results. GF simplifies the game creation process. With its sandbox IDE, a user can easily realize his/her own game ideas without much programming efforts.

3 GF Engine Design Approaches

The currently available game platforms in the market fall into two categories: One that aims for the best game quality, in term of scale and performance. One that is very easy to use, with limited flexibility.

In order to achieve best performances, the former platforms focus on every single detail of the gaming hardware to achieve optimal resource utilization. They usually come with libraries that provide access to alter hardware details down to the bits level. For example, Killzone engine (Valient 2007) allows direct communication with the gaming hardware so that programmers can carefully design their codes, maximizing memory access speed, and number of cache hits. Unfortunately, these skills require advanced knowledge and steep learning curve.

On the other hand, the latter type of engines, which are targeting casual users, have their own downsides. For example, Little Big Planet (Media Molecule 2008) allows gamers to create their own games, but with a lot of restrictions. Users are only allowed to build their games using a set of predefined game logics. The only full flexibility that the users can enjoy is to organize their game objects freely in the game world.

GF aims to strike a balance between the two; emphasizing on the ease-of-use without much compromise on the game quality.

3.1 Simplified User Interface

GF provides a simple yet effective user interface (Fig. 1). It comprises three main components:

Game Area: The window where users load and play their games. In GF, the game is always running. Users can seamlessly add or remove game logics without the need of restarting the game.

Script Editor: Provides facilities for users to define their game logics.

Fig. 1 GF user interface

Object Inspector: Provides facilities to inspect and modify the current game state (including attributes that are not visually displayed in the game area).

GF maintains high integrity among the three components. Each is linked to the others. For example, every line of script that creates an actor will be linked with the corresponding actor in the Game Area. When users delete or modify that line, the corresponding actor will be updated accordingly. For every class defined in the script, there will be a button created in the Game Area to quickly populate the scene with objects of that class. When an object in the Game Area is selected, all its properties will be displayed in the Object Inspector. Users can view and even modify these properties. All changes made will be converted to script and inserted into the game editor.

With this interconnection between the three components of the interface, GF interface is highly streamlined and simplified. Three aspects of the game production are put into three parts of the interface: game definition, game testing, and game inspecting.

It is a challenge to design a simple yet intuitive Script Editor. To ease the scripting process, an editor needs to expose various features to its users. One common approach adopted by most editors is by means of buttons and drop-down menus. As a consequence, users may be overwhelmed with all these Graphical User Interface (GUI) elements. Instead, Script Editor adopts the principle of context sensitive content to maximize intuitiveness. Only functions that are available at the current context will be displayed. In addition to minimizing the number of available functions, the number of contexts that can be switched from the current one are maximized. At any one time, users are expected to work with a

block of game definition in the Script Editor. All possible options for that block should be neatly displayed. The most appropriate location to prompt these options is at the text cursor, where they can be immediately applied. This design decision leads to two benefits:

First, text cursor is an intuitive indicator for users to make changes to the current block of game definition.

Second, in traditional approaches, dialog boxes will distract the users to certain degree. In GF, selectable options are both displayed and applied at the cursor location, keeping users' attention in one place, thus promoting faster scripting.

3.2 Simplified Work Flow

The contemporary work flow to create a game is complicated. The game ideas come from game designers and the artworks come from the artists. These requirements are communicated to programmers to produce the actual game. It is common to have miscommunication in the process, resulting in constant changes and tweaks being made until the game is finished. Furthermore, current game engines have to compile game codes written in a programming language to machine language before execution. The game needs to be restarted each time tweaks and changes are made, resulting in significant amount of time being wasted. These include the time needed to start the game (usually longer in the debug build), and the time to bring the game state to where it can show the changes (i.e., if the new change involves more game characters, the tester needs to recreate them to see the change). Also it would be useful for game designers to try out and see their ideas instantly. To simplify the work flow, GF reduces:

Communication time: GF enables game designers to directly create games. GF Factory emulates the role of programmers and translates game ideas to executable codes. This is achieved through the user interface and Rule-Oriented Simulation Architecture Language (ROSAL) language, which is based on Aspect-Oriented Programming (AOP). AOP itself is an extension of Object-Oriented Programming (OOP), see Sect. 5.3.

Time needed to verify the game changes: GF implements a feature, namely Live Sandbox, which enables the game to stay running and changes are instantly applied without the need of restarting the game. As a result, GF becomes a WYSIWYG (what you see is what you get) game editor. This also helps maintaining the game designers' stream of thought flow.

4 GF Architecture

This section describes the GF architecture. It also discusses the design decisions made to overcome challenges encountered and achieve the project objectives.

4.1 Components and Abstraction Layers Isolation

GF has a modular architecture (Fig. 2) allowing the replacement of one or all components it is built upon. In order to do this, GF needs to isolate the components it uses while still be able to extract the component's functionality. Currently, GF has three isolated components: rendering engine OGRE (The OGRE Team 2012), physics simulation engine NVidia PhysX (NVidia 2012b) and input engine OIS (Wrecked Games 2012). Furthermore, GF has two layers of abstraction, core engine layer and template-specific layer which are isolated as well. This section discusses the design to achieve these requirements.

4.2 Unified Data Structure

The first challenge comes from the different data representations of the components for the same thing. For example, the rendering engine uses (*x, y ,z, w*) for Quaternion, while the physics engine uses (*w, x, y, z*). These low-level data are

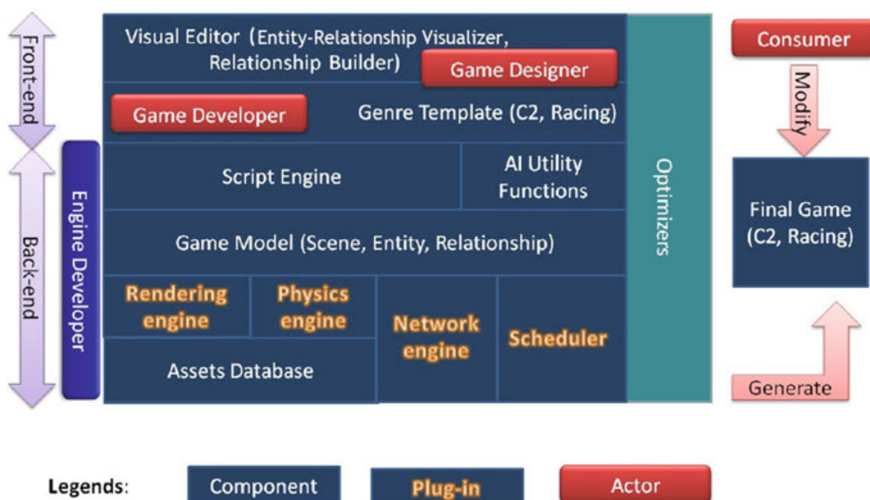

Fig. 2 GF architecture

exchanged between components at a very high rate for every frame and thus an efficient method is essential for data conversion and exchange.

Object transformation is usually represented by a 4×4 matrix for translation, rotation, scaling, and some tweaking like skewing. After some analysis, the matrix representation is inappropriate due to the following reasons:

The physics representation of object neither uses scaling nor skewing, only translation and rotation are considered. It is logical since object's size and shape are not changed in a physical environment. In some cases, such as skinned objects or soft-body objects, those changes are represented at a lower level.

The matrix size is too large for low-level data exchange, which happens at very high rate every frame. We use only translation and rotation using quaternion, i.e., two 4D vectors, which also save half of the exchange bandwidth.

In many cases, the matrix representation is decomposed back to translation vector and quaternion for computation. Thus, involving matrix will add up computation cycles to the data exchange.

Due to the above reasons, we pick the translation vector and quaternion as the basic data types between the rendering and physics engines. The next step is to solve the data type representation between them. Here are several approaches:

- Write conversion macros between the two corresponding types. For example, Vector3_to_NxVec3 and NxVec3_to_Vector3. This is the most efficient way in terms of performance, but is harder to comprehend for developers.
- Write an intermediate type which has automatic conversion operators to corresponding types. For example, we write class Vec3 which can be used in place of both Vector3 and NxVec3. This is the most elegant way for programmer while introducing much more CPU cycles for conversion.

We balance between the two approaches by introducing an intermediate class, derived from one engine, which has automatic conversion to the other type. Using the second approach saves half of the bandwidth. In the case of changing engines, the two intermediate classes (Vec3 and GFQuad) need to be rewritten.

4.3 Conflict of Data Organization

Different components have different ways to organize their data structure. The OGRE rendering engine uses local coordinate system to organize objects while the PhysX engine uses global coordinates. Furthermore, each object in OGRE must be given a unique name, which is quite inconvenient to instantiate an object. To resolve the coordinate conflict, several things are need to be considered:

- The direction of data exchange.
- The coordinate system we use.
- How to maintain features in hierarchical coordinate system.

4.4 Resolving Runtime Data Type Checking

In order to trap physics events, such as collision or ray cast hit, the physics engine provides a pointer to store *userdata* to identify which object is responsible for the event. It is not possible to get the type information of these objects from the event callback without carefully designing the class structure. Since GF uses C++, which does not store dynamic type information like other advanced languages, it can only store one single type in the *userdata* of the physical objects. This type will be responsible for implementing the event handlers. Polymorphism will play the role of type identification. There are multiple options for this type selection:

Store the Physical Object. This approach hides the physics component interface although we need to store a pointer to GameEntity, since the entity is what we actually need.

Store the GameEntity object. This approach returns directly the object we need but expose dependencies on the physics component.

Identify the general physical events and make wrappers around them. GameEntity will adopt this wrapper as one of its parents. This is GF's choice since it has all the benefits of the other approaches.

Another issue is to support type checking at the script level, because ultimately, most of the physical events need to be accessible in the script space.

4.5 Template API Isolation from Core Engine

GF divides object declaration into two levels. At the core engine level, objects are defined with general purpose properties, while at the template level, objects are added with more specialized properties. For example, the C2 (Command and Control) Template (see Sect. 4.7) needs to add other types such as Selectable Object, Movable Object, Attacking Object that can be mixed with the core object types in a variety of ways.

In GF class design, we adopt multiple inheritances. Functionalities are implemented as classes, such as PhysicalEventHandler that is actually an interface with all pure methods, or ScriptID, which is a constructor wrapper to record the object address for object identity at script space. Classes like ScriptID are treated as self-implemented interfaces. Other classes, which want to possess these features, can be extended from these base classes, along with their own primary parent classes. Multiple inheritance enables GF to create combinations of features. This is especially useful at the template level, where many classes share a subset of features. For example, Tank is extended from AcquirableObject and SelectableObject; whereas Soldier is extended from SelectableObject and MovableObject, etc.

GF supports development of multiple types of game by providing a template to each game genre. The template is developed by a programmer and exposing facilities that a game designer can use for faster game development.

4.6 Core Engine Class Design

The class relationship for the core engine is illustrated in Fig. 3. The blue outlined classes are abstract classes. The red generalization lines are virtual inheritance.

The core engine classes are designed with the following wrapping principles:

Step 1 Identify main objects from the underlying engines. For example, the physics engine consists of rigid object, composite object, controllable object, vehicle objects, etc.

Step 2 Wrap these objects so that the final interface contains no dependency on these objects or the relationship between these objects

Step 3 Using the wrapped interface, connect objects from different engines together to form a final object which is visible at scripting level

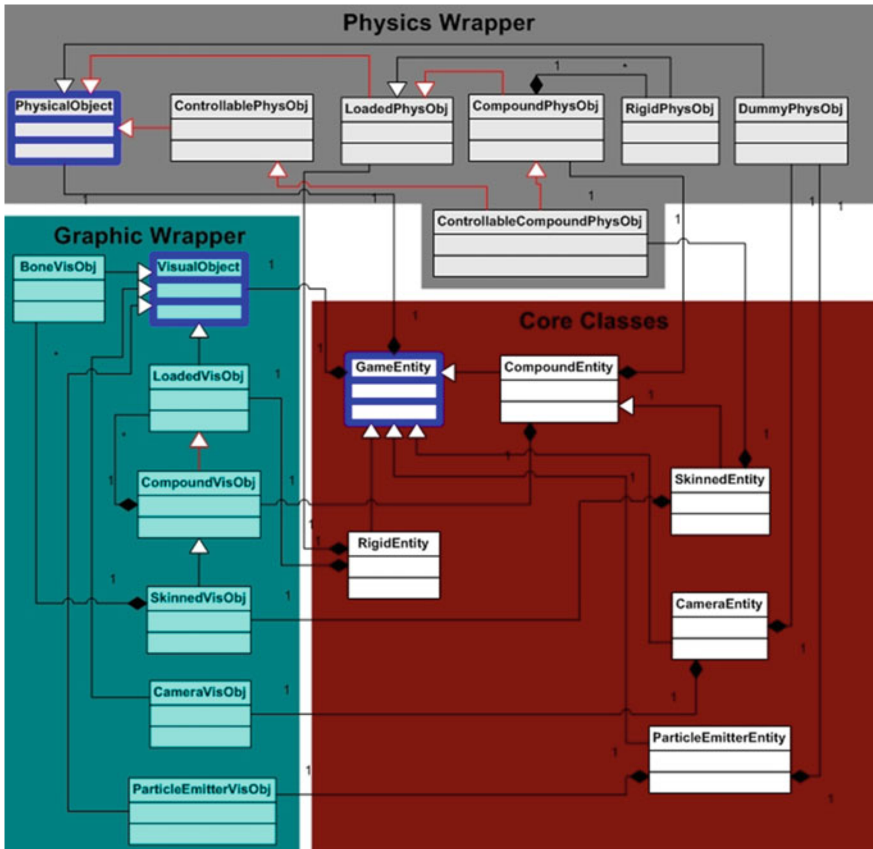

Fig. 3 Core engine class diagram

In Fig. 3, we identify the commonly used objects in Graphic Wrapper and Physics Wrapper. These objects can be found in other similar engines as well. Therefore, when a better engine is introduced, GF can simply adopt it by writing a new wrapper based on the existing interface.

4.7 Template Class Design: C2 Template

The logic within a template should derive only from the core classes to be independent of the underlying components. Furthermore, the class design should be flexible for a complex set of object with mixed behaviors. The current design is illustrated as in Fig. 3. The blue outlined classes are abstract classes. The cyan classes are self-implemented interfaces described in the previous section.

Every object in the C2 Template is derived from a single core class and a combination of self-implemented interfaces. One object can inherit features from these self-implemented interfaces by simply extending from them. Some interfaces work with their container, for instance, the SelectableObject is manipulated by the EntitySelector. These classes reflect our vision of a C2 game, where objects can be selected and commands can be issued to each object.

5 GF Engine Components

Figure 4 shows the various components in the architecture of GF. Due to space constraint, only the details of the rendering, physics, and script components are discussed in this section. For asset database, instead of using proprietary data format or tool set, GF adopts the emerging standard COLLADA (Khronos Group 2012) for all kind of digital assets. This allows the use of common digital content creation tools such as 3ds Max and Maya. Game artists can then make the best use of their experience with their favorite software without having to learn new ones. An example of artificial intelligence component can be found in (Wardhana et al. 2009).

5.1 Rendering Component

Besides the usual Forward Shading, GF rendering system also supports the Deferred Shading (Hargreaves 2004) technique. GF currently uses OGRE's features for rendering, and shading languages supporting Cg (NVidia 2012a), GLSL and HLSL.

Material Rendering GF supports material down to the shader level. The shader is written by graphics programmers and can be reused by artists.

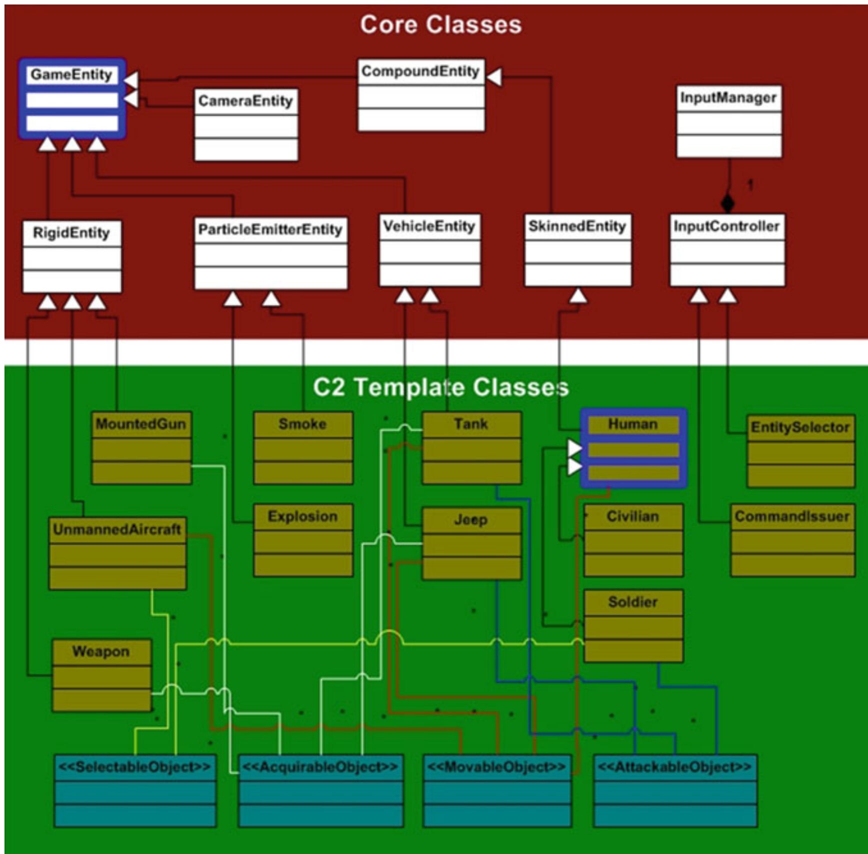

Fig. 4 C2 template class diagram

Compositor System GF has a complete system for full screen effect using OGRE compositor framework. These effects are achieved by applying shader materials to the whole screen, or render target texture. GF supports multiple rendering windows, which are incorporated into its GUI. Users can add more cameras and display these cameras by using these rendering windows.

Debug Renderer A debug renderer is supported for all physical bounding shapes, axes, and constraints. Objects can be switched to wire-frame or dot mode to clearly see hidden details (Fig. 5).

Deferred Shading Deferred Shading is a modern rendering technique leveraging the new rendering hardware platforms. It becomes more and more popular in the recent years. Many award winning games (e.g., Killzone 2, God of War 3, Little Big Planet, etc.) adopted the Deferred Shading technique and successfully created the best real-time graphical presentation up to now.

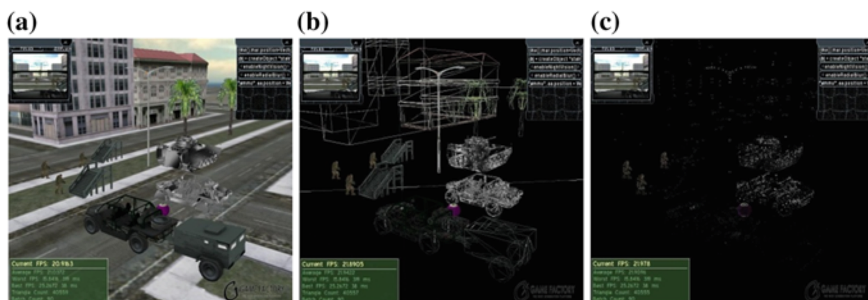

Fig. 5 Debug renderer **a** Fully rendered mode **b** Wire frame mode **c** Dot mode

The strongest aspect of Deferred Shading is the low cost of rendering a light. A traditional rendering engine usually limits the number of lights that simultaneously affect an object. The limitation ranges from 4 to 8. If this limitation is exceeded, one or more lights will be disabled. In the case of dynamic lighting, going in and out of the limitation may result in a blinking scene. This usually leads to a strict lighting design that limits the number of dynamic lights in the entire scene to 4 or 8. By introducing low cost lighting, Deferred Shading provides flexibility in lighting design for the scenery and even game play.

Furthermore, by shifting lighting computation to the end of the rendering passes, shadow manipulation becomes much easier. Shadow casting lights can now share a common shadow map texture, which is not possible in traditional rendering engine. This eliminates memory constraint imposed in shadow casting lights, allowing a larger number of shadow casting lights in the scene.

With the availability of depth surface in Deferred Shading, light can be modified to be used in various circumstances such as decal or caustic. Grouping objects into different groups, z-sorting can be applied to these forms of lights. For example, static decal will be applied only on static objects and dynamic objects can obscure it; whereas dynamic decal will cast on all objects.

Special Effects Deferred Shading renders all objects into three flat surfaces: normal surface, depth surface and color surface. These surfaces provide rich information for a lot of effects, such as Depth of Field and Screen Space Ambient Occlusion (Fig. 6). Furthermore, the cost of such effects is reduced significantly as each pixel is computed only once for the effects.

5.2 Physics Component

The current physics simulation in GF is NVidia PhysX. This is a close source engine with license to use freely for commercial purposes. So far, this is the only engine with hardware acceleration, supported by NVidia, the flagship company that produces graphics cards, where some of them have PhysX hardware support.

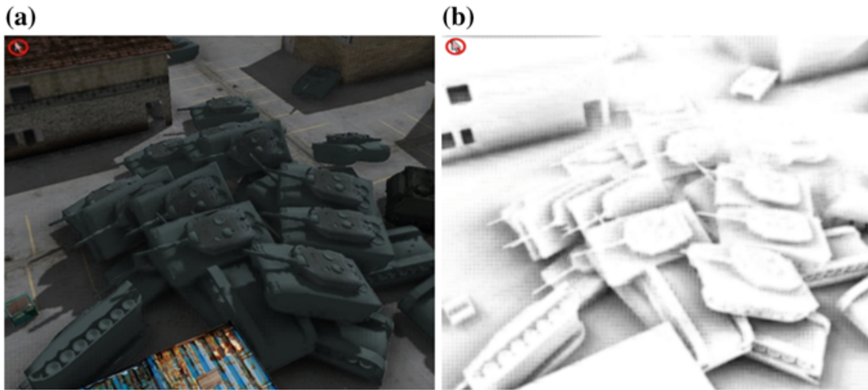

Fig. 6 Screen space ambient occlusion **a** Fully rendered frame **b** Screen space ambient occlusion component

Rigid Body Simulation and Collision Detection Rigid body is an object that keeps its size and shape unchanged after the collision, i.e., an unbreakable object. Rigid Body simulation is performed extremely fast in PhysX with Broad Phase.

Collision Detection and multicore support. GF supports loading complex data structure to represent the Rigid Body shape. PhysX supports from basic shapes with low cost in collision detection (such as box, sphere, capsule, cylinder, etc.) to advanced structures (such as height field, convex mesh, etc.). GF supports ray casting for multiple uses. Ray Casting is a collision detection manner using a ray casted from an origin to infinity. This is performed really fast and allows a large numbers of rays in one frame. Ray Casting is used in mouse pickup and way-point generation.

Physical Constraint Physical constraint links objects together in a specific way. GF supports multiple types of physical constraints, from hinge joint, sphere joint, sliding joint to 6 degrees-of-freedom (DOF) joint. One of the applications of physical constraint is Rag Doll physics. GF has implemented high quality Rag Doll physics for all skinned objects (human, animal or other biological entities). This information of Rag Doll can be loaded from files created with digital content creation tools, such as 3ds Max. The visual and physical information is linked using name matching (in this case, the bounding shapes have the same names with the corresponding bones).

GF also supports breakable constraints. The constraint can be given a limit on the force that it can sustain. If the force exceeds this limit, the two connected physical objects will be disjointed. This is very useful to model destructible objects, such as house and car.

Controllable Object GF uses a physics simulation for all objects inside the game. Therefore, it is hard to directly control an object movement using only physical input, such as force and torque. These are the reasons:

It is hard to make object stand still. We can prevent the sliding of the object by setting a high friction value. However, this approach may make the object hard to

be moved, or move inconsistently on different surfaces. It is hard to make object move without rotating since any force deviated from object's center of mass will result in a rotation. It is hard to make object climb up small obstacles. It is hard to make object continue its movement when collides with obstacles.

GF also has designed a controllable class that solves all the problems mentioned above. We use a capsule shape for the object and handle all collision events manually. Furthermore, GF allows switching from controllable state to any other physical states. For example, the soldier can turn from controllable physical state into compound physical state (for Rag Doll death animation).

5.3 Script Component

GF exposes its features to game designers through a scripting language. A set of selectable sections of the script can be exported to the end-users (i.e., gamers) for further modification. GF has chosen script, instead of pure GUI approach, for more flexibility and scalability. In order to meet the requirements of GF, the script subsystem needs to provide:

A supporting IDE with easy-to-use interface for rapid game development.

A scripting language, instead of pure GUI approach, for finer adjustment. The language should conceal programming-specific details with its advanced features.

Runtime performance of the final game should not be compromised.

Support classic programming paradigms, such as Object-Oriented or Procedural, to be friendlier to programmers.

ROSAL Design Principles Previous game engines are targeting programmers as their users. Therefore, the integrated languages of these engines usually focus on Object-Oriented Programming (OOP) design, which is very familiar to programmers and allow construction of complex game structures. Although being the most popular programming paradigm at the moment, an OOP language suffers from high maintenance cost due to cross-cutting functionalities which span over the boundaries of class encapsulation (Constantinides and Skotiniotis 2002). This problem is solved in Aspect-Oriented Programming (AOP) (Kiczales et al. 1997) language, which is an extension of OOP and separating cross-cutting functionalities into individual concerns. However, AOP languages are not popular due to its complexity.

By changing the target users to game designers and gamers, GF scripting language needs a different approach. ROSAL is designed around the concept of AOP which has enough modularity for simulation application while still being an easy to use language.

ROSAL has a number of features supporting the reading and maintenance of the code. First, ROSAL has a syntax resembling human language. Second, its IDE can display explanatory information for every piece of the code. Third, the IDE Diagram Generator can help to illustrate complex architecture and relationship between game entities. Finally, the IDE Filter provides quick information retrieval and code organization.

ROSAL, as its name suggests, focuses on the construction of rules. The programmer just declares the rule in a way that is very similar to its real-life counterpart. The language takes care of how the rules are implemented.

ROSAL is a multiparadigm programming language. It is based on OOP and thus supports inheritance, polymorphism, and encapsulation. Furthermore, it has separation of cross-cutting concerns borrowed from AOP. This helps in organizing the program structure. It is built upon Lua (Ierusalimschy et al. 2007) and Metalua (Fleutot 2007). Lua is an effective scripting language with high customization and its semantics can be changed using well-defined meta-methods, while Metalua is a Lua parser generator written in Lua. In addition, ROSAL also encompasses a C++ Luabind (Wallin and Norvig 2003) mechanism in which C++ objects could be extended via a scripting manner, which allows rule-based scripting functionalities to act on the objects. With ROSAL in-conjunction with C++, we aim to provide developers better choices as compared to the more general purpose programming languages Actionscript (Crawford and Boese 2006), C#, Python, Java, etc. This coding choice enables more compute intensive and hardware-related tasks to be carried out in C++ while allowing the rules to be scripted in ROSAL in a more intuitive manner.

The language supports both imperative and declarative programming. When describing a rule, a programmer just declares which objects are involved by defining a condition. The language will determine how to retrieve these objects in a CPU-saving manner. After the objects are identified, a sequence of instructions is carried out. By doing this, the language can work totally at class level, which means the programmer is relieved from the burden of effectively storing and retrieving objects.

6 Application and Coding Example

Through a typical game production pipeline, we illustrate the use of GF to develop a 3D computer game. GF acts as a reusable library that provides objects and functionalities, as well as adding new functionalities and gaming components.

6.1 Game Production Pipeline

Figure 7 shows a typical game production pipeline. The production process consists of designing game resources (objects and characters, their animation), and then importing them into GF before performing the necessary programming and scripting to implement the gameplay with respect to the game story.

Design of Game Resources The game resources consist substantially of (but not limited to) the game objects, characters, their animations and the necessary texturing materials. These 3D characters, comprising their geometry, physical attributes, animation, and materials, can be designed and created by common practices such as

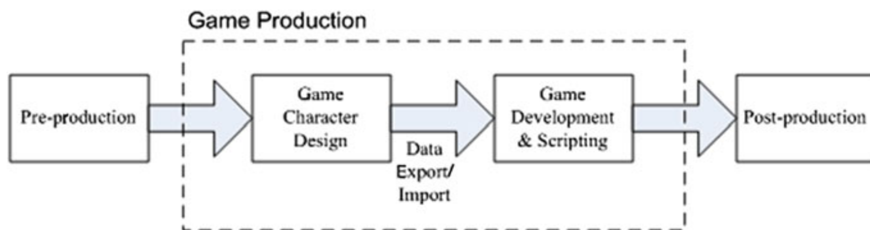

Fig. 7 Typical game production pipeline

via using of software tools like 3ds Max. These resources are exported into compatible formats that will be loaded into the game engine.

Types of Game Objects and Characters There are several basic types of objects and characters in GF. We defined them as entities: static rigid entity, movable rigid entity, skinned entity, and interactive entity. The characteristics of these entities are decided during the design and creation phase. These entities are then inherited so that their fundamental features are further extended. For example, an animated soldier character is inherited from a skinned entity to allow the surface mesh model to deform according to its underlying skeleton joints.

Programming and Scripting Coding in GF using C++ allows extensions to the general functionalities, objects, graphics and physics algorithms, and be exposed in a form that is callable by the rule-based ROSAL scripting that allows non-expert developers to simply code real-time discrete game elements and events.

6.2 First-Person Shooting Game Example: C2 Game

This is a first-person shooting military-based game, which makes use of the graphics, physics and audio components of GF, as well as extending the engine's core functionalities. Scene and object creation are done through 3ds Max (Fig. 8) with NVidia PhysX plug-in that enables the handling of dynamics and collisions.

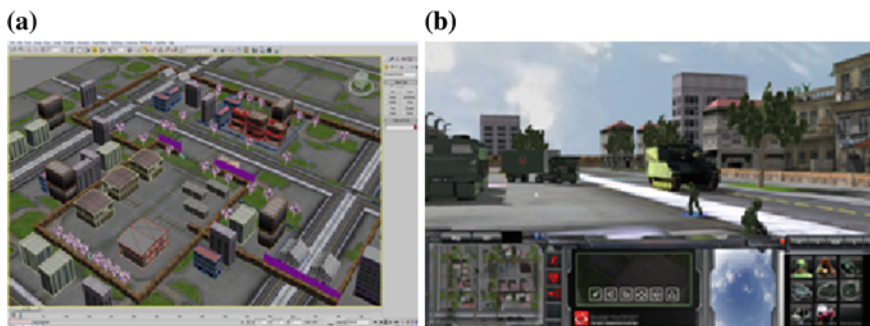

Fig. 8 First-person shooting game **a** Scene created in 3ds Max **b** Scene imported into GF

The fundamental object entities of GF are extended into more specific entities, such as the soldier and vehicles, coded in C++ for computing effectiveness:

```
// inherit Soldier from SkinnedEntity
class Soldier : public SkinnedEntity {
   public:
      virtual void FrameMove();
      // extending the each frame event
      ...
      void TakeCover();
      // new functionalities for Soldier
      void AttackEnemy();
      ...
};

// inherit Vehicle from MovableRigidEntity
class Vehicle : public MovableRigidEntity {
   public:
      virtual void FrameMove();
      // extending the each frame event
      ...
      void Steer();
      // new functionalities for Vehicle
      void Brake();
      ...
};
```

The main step for creating the game-play is loading of scene and characters via scripting:

```
-- to load scene
Scene.loadScene('MilitaryPrototype')

-- create extension soldier class in ROSAL
inherit Marine from Soldier

-- setting attribute of a marine
Marine.HealthPoint = 100
Marine.WhichTeam = 'A'
Marine.Ammo = 10000
...

-- create an instant of Marine type
mm[1] = Marine()
...

-- extending other gaming entities/characters
inherit Jeep from Vehicle
inherit Tank from Vehicle
...
```

Controlling the game entities

```
if key == 'W' then
  -- animation from the resource in response to key 'W'
   _getBaseObj(Marine):animation('walk_forward')

if key == 'S' then
  -- animation from the resource in response to key 'S'
   _getBaseObj(Marine):animation('walk_backward')
...
```

Rules that operate among the game entities

```
rule 'marine attack enemy'
   actor marine as Marine, enemy as Marine
   when distance(marine, enemy) < 30
      and marine.HealthPoint > 0.5
         marine.OpenFire()
   end
end

rule 'marine flee'
   actor marine as Marine, enemy as Marine
   when distance(marine, enemy) < 30
      and marine.HealthPoint > 0.5
         marine.Flee()
   end
end
```

7 Conclusion

Many software development paradigms and methodologies serve a common purpose: to address the myriad forms of inefficiency in the software development process and the brainchild of any such methodology originates typically from practical experience. However, the solution of adopting all practices of a methodology may be difficult in the case of game development.

To date, many game engines abstract production artifacts as data (content), behavior/logic and the runtime (engine). Redeployment of software for a similar game genre is superficially achieved through either modification of data layer or "rewiring" of the code and this typically requires the user to have sound knowledge of the game engine's technology. One of the key emphases of GF's design is reusability which is achieved through custom application templates. Such a design brings forth the advantage of accelerating the generation of common features while isolating the user from the low level implementation details. Under the circumstance when development is necessary for a specialized need, such effort can be "amortized" over future reuse scenarios through templates which are more efficient and manageable than raw code manipulation.

Another innovation of GF's overall design is its ability to stay "future-proof". Through software frameworks, only high level adaptors need to be built for

specific functions that pertain to a certain genre of application. The implementation details will be totally abstracted from the user such that as the technology of a particular field advances, the system can remain updated by integrating the best-of-breed components into it. This approach contrasts greatly from the monolithic design of many proprietary game engines whereby the integration of auxiliary components may be difficult.

While the GF framework is functional and operational, many desirable features such as automated terrain generation, dynamic generation of destroyable objects, advanced artificial intelligence for non-player character, networking features to support MMOG, and expanding GF to run on diverse platforms are missing. These are in our future R&D plan.

Acknowledgments We would like to acknowledge the support by the Singapore Defence Science and Technology Agency under Project Sigma and contributions of Gabriyel Wong, Hai Nam Pham and Yin Mun Wong during the early development of GF, which was known as Game Factory then.

References

Constantinides C, Skotiniotis T (2002) Reasoning about a classification of cross-cutting concerns in object-oriented systems. In: Costanza P, Kniesel G, Mehner K, Pulvermüller E, Speck A (eds) Second workshop on aspect-oriented software development of the German information society

Crawford S, Boese E (2006) Actionscript: A Gentle Introduction to Programming. J Comput Sci Coll 21(3):156–168

Fleutot F (2007) Metalua manual. http://metalua.luaforge.net/metalua-manual.html. Accessed 12 June 2012

Hargreaves S (2004) Deferred shading. Game developers conference. http://ambility.com/DeferredShading.pdf. Accessed 12 June 2012

Ierusalimschy R, de Figueiredo LH, Celes W (2007) The evolution of Lua. In: Proceedings of the third ACM SIGPLAN conference on history of programming languages, pp 2-1–2-26.

Khronos Group (2012) COLLADA - 3D Asset exchange schema. http://www.khronos.org/collada/. Accessed 12 June 2012

Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J (1997) Aspect-oriented programming. In: Aksit M, Matsuoka S (eds) ECOOP'97 object-oriented programming, lecture notes in computer science, vol 1241. Springer, Berlin, pp 220–242

Media Molecule (2008) Little big planet. http://www.littlebigplanet.com/. Accessed 12 June 2012

NVidia (2012a) Cg Tutorial. http://developer.nvidia.com/node/76. Accessed 12 June 2012

NVidia (2012b) PhysX. http://www.geforce.com/hardware/technology/physx. Accessed 12 June 2012

The OGRE team (2012) OGRE. http://www.ogre3d.org/. Accessed 12 June 2012

Valient M (2007) Deferred rendering in Killzone 2. http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf. Accessed 12 June 2012

Wallin D, Norvig A (2003) Luabind. http://www.rasterbar.com/products/luabind/docs.html. Accessed 12 June 2012

Wrecked games (2012) Object oriented input system. http://www.wreckedgames.com/forum/index.php/board,6.0.html. Accessed 12 June 2012

Wardhana NM, Johan H, Loh PKK, Seah HS, Ong DWS (2009) An efficient connection graph for waypoints in virtual environments. In: Proceedings of international conference on computer games, multimedia and allied technology (CGAT), 273–282