MoonGate: RTS engine with user-oriented architecture

Rudolf Kajan, Adam Herout
Department of Computer Graphics and Multimedia
Faculty of Information Technology, Brno University of Technology
Brno, Czech Republic
ikajanr@fit.vutbr.cz, herout@fit.vutbr.cz

Abstract

This paper introduces MoonGate - a real-time strategy engine based on replaceable components, thus serving as an easily customizable educational environment for studying a wide range of problems in several research areas. The implemented engine also supports, alongside the Windows platform, Microsoft's Xbox 360 gaming console, making it the first open source starter kit and real-time strategy engine for Xbox 360. MoonGate's main goal is to attract students, academics and indie game developers to learn by creating highly interactive and visually attractive games, and by providing easily understandable and highly customizable educational environment which incorporates the best features of modern commercial games, visualization environments and game middleware.

1. Introduction

Out of many game genres that are available nowadays, one of the most important is the genre of real-time strategy games, since they are useful from an entertainment as well as serious games perspective [1]. Researchers such as [2], [3], [4] or [5] suggest that real-time strategy games also offer a wide variety of fundamental AI research problems.

Real-time strategy is the genre of computer games that encompasses tactical games that happen in real-time. "Real-time" means that there is a continuous flow of time in the game world, so that immediate decision making and responding quickly to arising situations is important. These games can be viewed as simplified military simulations played by two or more players where each player can be in control of potentially hundreds of units with specific properties and abilities.

One of the current problems is that commercial game developers and researchers do not focus on exactly the same elements. Up until now, game companies have spent more time on improving the game's graphics more than any other part of it and commercial RTS games are closed software which prevents researchers, students and hobbyists from connecting their own modules to them. A shortage of artificial intelligence competitions based on real-time strategies also deprives artificial intelligence researchers and enthusiasts of an opportunity to compare their algorithms [6]. Moreover, available free RTS engines are usually too complex to serve as a practical educational and demonstrational tool in courses (for example, Spring [10] engine v.0.8.1 has 340641 lines of code in 1425 files).

In order to address these problems, *MoonGate* [7] – an open source, real-time strategy game engine for Xbox 360 and Windows platforms, was designed and implemented. *MoonGate*'s main purpose is to attract academics, students and hobbyists to learn by creating games and giving them an opportunity to easily study various problems by providing easily understandable and customizable educational environment accompanied by a repository of components varying from terrain, particle systems, bone animations to shaders. These are easily readable and reusable without the need to study the non-related support parts of the code as it is common in other engines.

MoonGate Engine was designed and implemented with the following in mind:

- Simple extensibility through usage of reusable components.
- Content pipeline that allows easy creation and import of content.
- Tweaking without the need of source code recompilation.
- Exploiting benefits of target platforms for learning purposes.

In order to achieve the mentioned properties, the XNA Framework and the Xbox 360 game console were chosen as the most suitable to create a base upon which *MoonGate* was implemented.

One of the reasons behind the selection of the XNA Framework is the fact that since its release in 2006 XNA has seen a surge of momentum with more than 1 million downloads of the tool and more than one thousand academic universities globally [8] are using XNA Framework in their classrooms.

That is why *MoonGate* can stay really close to its target audience – students, teachers, and indie game developers and keep constantly evolving.

2. Existing real-time strategy engines

Currently there are only a few open source real-time strategy engines freely available to the general public that were developed to a phase when they are actually usable as a visualization tool or a test-bed for algorithms. The vast majority of produced engines are commercial, which means only very limited usability by students and researchers due to severely limited options in their modification. Some of the mostly used 3D open source real-time strategy engines are stated below.

2.1. ORTS – Open Real Time Strategy

The ORTS project [9] was started with the goal of creating a free software system that lets people and machines play fair RTS games. It is a programming environment for studying real-time AI problems such as path finding, dealing with imperfect information, scheduling, and planning in the domain of real-time strategy games.

The communication protocol is public and all source code and artwork is freely available. Users can connect whatever client software they like. This is made possible by a server / client architecture in which only the currently visible parts of the game state are sent to the players.

This openness leads to new and interesting possibilities ranging from on-line tournaments of autonomous AI players to gauge their playing strength to hybrid systems in which human players use sophisticated GUIs which allow them to delegate tasks to AI helper modules for increased performance.

ORTS is not a single RTS game, but an RTS game engine. Users define the game they want to play in the form of scripts which describe all unit types, structures and their interactions. These scripts are loaded by the ORTS server and executed.

The second part of the ORTS system is client software which connects to the server and generates actions for objects in the game. The server sends player views to the clients and receives actions for all the player objects, which are then executed. This loop is executed multiple times a second. If the 3D graphics client is connected, the world is rendered using OpenGL and the user can issue commands using the mouse and keyboard.

2.2. Spring engine

Spring [10] is a project aimed to create a new and versatile open source (GPLv2) real-time strategy engine. Spring is a multi-platform project written in C++, using OpenGL, OpenAL, SDL, boost, 7zip and Lua scripting language.

At the moment Spring fully supports Windows and Linux platforms, while the Mac OSX version is currently not available. Spring is well-known for massive battles limited only by the power of the host computer – up to 30,000 units and up to 250 players can fight on a single map. Fully featured lobby clients allow to easily play multiplayer games. These clients have built-in support for all standard features like automatic game and map downloading, chat and friends lists.

Spring's architecture allows for the usage of third party AIs that can work very competently in many cases with multiple Spring engine based games.

Very extensive Lua interface allows users to create custom (graphical) user interfaces. Through third party widgets, users are able to improve not only the GUI, but also gameplay. Spring with a built-in physics engine supports realistic projectiles trajectories, deformable terrain, forest fires and dynamic water.

2.3. Glest

Glest [11] is a free 3D real-time strategy game built on a custom engine. It is a multi-platform project written in C++ with portability in mind, so it can be compiled easily on Windows, Linux, FreeBSD and Mac OSX. Glest uses the cross platform OpenGL API to render 3D graphics. For unit models and animations, Glest uses its own 3D format; an export plugin for 3D Studio Max, and tools for Blender are available. Every unit, building, upgrade, faction, resource and all their properties and commands are defined in XML files.

2.4. Common problems

The problem that often plagues (mainly freely available) game engines and starter kits is difficult extension. These engines and starter kits are not designed with extensibility in mind from the beginning. When users would like to extend them in some way, it is often almost impossible to do so without making (rather large) changes in overall project structure, mainly due to changes not only in components, but also due to changes in connections among these affected components. Afterwards, thorough regression testing is usually needed to verify the correctness of changed couplings and parts of the code. These engines and starter kits are thus hardly

usable – easy extensibility is one of the key features that are expected.

Some engines even use content that is very hard to understand and create. For example, we can see quite often binary 3D models with built-in parameters for an engine's model and texture processors that were set in 3D content creation application, shaders that are referenced directly from models or skeletal animations exported in an undocumented way. In this case, it is difficult to create one's own game content, which again lowers the usability of an engine or a starter kit as a whole.

It is not an easy task to avoid these common pitfalls and create an educational environment that is not overwhelmingly complex and easily customizable. There are examples of successful projects in nearly every game genre, like the already mentioned Spring from real-time strategy genre, which is often chosen for visualization of AI algorithms, or genre of role-playing games, where many solid educational videogames have been developed to run on one of the iterations of Neverwinter Nights [14] using the Aurora Neverwinter Toolset (Aurora is part of Neverwinter Night's installation). Many of these have been designed by teachers for their classrooms and released to the general public.

If we look at these projects, regardless of what genre they belong to, we can clearly see that these engines are completely modifiable, thus making it fairly easy to manipulate for desired educational outcomes. Although they present a full 3D virtual interactive environment, complete with anthropomorphically correct characters, their runtime requirements are relatively light. These features are basic principles which *MoonGate* follows as closely as possible.

3. Main design objectives

Although the *MoonGate Engine* was originally designed as a starter kit for real-time strategy games for Microsoft's Xbox 360 gaming console that would allow for fast creation of these games in an easy and convenient way, positive feedback from its users soon showed that it can also be a valuable tool not only for indie game developers, but also for students interested in game development for PCs and Xbox 360 game console.

There are already companies exploiting the console's hardware in order to create edutainment software, as for example Educomp Group, which is an education solutions provider and the largest education company in India that reaches out to over 25,000 schools. They chose Xbox 360 as a platform to promote its interactive learning programme [12], but these companies focus mainly on children attending elementary schools and create only simple educational games, rather than focusing on students attending university courses and people with game development as a hobby.

On the other hand, *MoonGate*'s purpose is also not to directly compete with projects like ORTS that focus solely and deeply on a single selected problem which, for example, in ORTS's case is studying real-time AI problems. Of course, because *MoonGate* is a RTS engine, it is able to serve as a visualization tool for areas within AI research, as for example:

- **Resource management**. Resource management is a vital part of every strategy. Players must immediately create a supply chain in order to start producing defense and attack forces and structures, climb up the technology tree and purchase upgrades for units.

- **Decision making under uncertainty**. At the beginning, game players are not aware of the enemies' locations of buildings, units or places that are being used to harvest resources from. Each unit has a certain radius in which it is able to "see" other units and it is up to the player to use units and their combinations in order to obtain intelligence as quickly and accurately as possible.

- **Spatial and temporal reasoning**. Static and dynamic terrain analysis as well as understanding temporal relations of actions is of the utmost importance in RTS games – and yet, current game AIs largely ignore these issues and fall victim to simple common-sense reasoning [13].

- **Collaboration**. In real-time strategy games, communication and coordination of actions among groups of units and players is a must.

- **Opponent modeling and learning**. AI opponents in most contemporary real-time strategy games have a great handicap when compared to human players – they are not learning from experience.

But what is important, *MoonGate* and RTS games in general are also very useful as a test bed and visualization environment in many other areas like rendering of a large terrain, particle systems, bone systems and many others.

MoonGate aspires to act as a bridge between commercial games, visualization environments and game middleware by providing an open source, easily customizable environment for studying a wide range of problems from several research areas.

4. System overview

MoonGate Engine was designed from the beginning as a lightweight environment that tries to be as generic and flexible as possible. It provides a large amount of functionality right out of the box, but allows for quick and simple modification of each and every part.

MoonGate Engine uses the XNA Framework as the lowest layer. Its purpose is to provide the most basic platform abstraction and access to hardware.

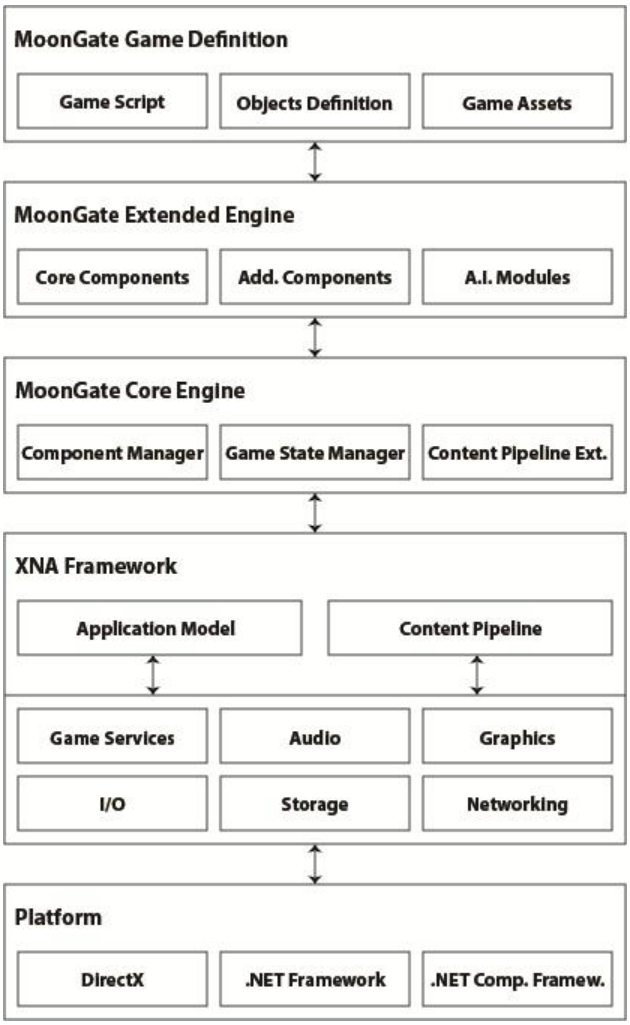

Figure 1. Architecture of MoonGate Engine with its three layers – Game Definition, Extended Engine and Core Engine built on top of the XNA Framework. Arrows denote communication between layers

MoonGate Engine consists of three main parts - *MoonGate Core Engine*, *MoonGate Extended Engine* and *MoonGate Game Definition* (See Figure 1), which form layers where each layer takes functionality of the lower layer, extends it and brings more abstraction. These layers have the following purpose.

4.1. MoonGate Core Engine

MoonGate Core Engine is built directly on top of the XNA Framework. It uses its access to hardware and basic application model along with content pipeline and extends them to create a foundation for upper layers which are, unlike *MoonGate Core Engine*, game-genre specific.

MoonGate Core Engine itself is not designed specifically for real-time strategies, but provides a rather generic foundation for different types of games. This is achieved through custom content pipeline extensions, generic game state manager and a component manager used for components registration.

Custom content pipeline extensions currently support the most frequently used tools and 3D models, textures, fonts and audio formats. These extensions consist of content importers and content processors which allow for the usage of various graphic effects and file formats unsupported directly by the XNA Framework. They are commonly used in modern computer games, as for example in the creation of mipmaps or the usage of various mapping techniques (e.g., normal mapping).

MoonGate Core Engine also contains a generic game state manager responsible for maintaining the stack of one or more *GameScreen* instances. It coordinates the transitions from one screen to another, and takes care of routing user input to whatever screen is on top of the stack. Each screen class, including the actual gameplay which is just another screen, derives from *GameScreen*. This provides methods for updating, drawing and input handling, along with logic for managing the transition state. *GameScreen* does not actually implement any transition rendering effects; however, it merely provides information about transition progress, leaving it up to the derived screen classes to process the information in their drawing code. This makes it easy for screens to implement different visual effects on top of the same underlying transition infrastructure.

One of *MoonGate*'s main advantages is the usage of reusable components – *GameComponents* which are registered, initialized and managed by the *Component Manager*. The idea behind *GameComponents* is to provide a way to keep a reusable code in a nicely encapsulated form. The dependencies between classes can also be managed and de-coupled from one another, so that they can be removed, altered or replaced in isolation. They can be even dropped into other completely new projects quickly and easily. Components that need to access each other do so only through well-defined interfaces that could be provided by a different component in another game, or not at all. *GameServices* are a mechanism for maintaining a loose coupling between

objects that need to interact with each other. Services work through a mediator—in this case, *GameServices*. Service providers register with *GameServices* and service consumers request services from *GameServices*. This arrangement allows an object that requires a service to request the service without even knowing the name of the service provider.

For example, to register an object that provides a service represented by the interface *IMyService*, the following code would be used:

Services.AddService(typeof(IMyService), myobject);

4.2. MoonGate Extended Engine

MoonGate Extended Engine focuses mainly on functionality that is relevant mostly for the genre of real-time strategy games by providing a set of reusable components containing game logic and artificial intelligence modules which define the behavior of agents used in game.

This layer consists of *GameComponents*, which can be further divided into *Core Game Components* and *Additional Game Components*, and *A.I.Modules*.
Core Game Components are *Game Components* which provide basic functionality are essential for gameplay. An example can be a component responsible for rendering of terrain, an input management component or an object creation component.

Additional Game Components are optional components that are not essential for playing the game, as for example the in-game game state management console or a component showing the number of frames that are rendered every second. These components provide support for various tasks like performance monitoring, debugging or creation and editing of the game environment.

A.I. Modules refers to artificial intelligence algorithms used by agents in the environment. *MoonGate* provides several pathfinding and resource management algorithms along with classes describing different behavior of agents out of the box as starting examples. It also encourages users to import and test their own algorithms.

4.3. MoonGate Game Definition

MoonGate Game Definition layer contains XML-based configuration files. These files describe the game and define all static and dynamic in-game objects along with their representation and properties (for more information about *Objects Definitions* see Section 5.3).

When running an application based on the *MoonGate Engine*, clients - PCs and/or Xbox 360s are using a peer-to-peer network topology to exchange data during network sessions (see Figure 2).

One of the clients is always the session host. This computer is responsible for the execution of the *Game Script* – script that defines the game that will be played along with precise terrain definition and initial placement of objects on the terrain. Each client afterwards loads *Game Assets* – in-game models, textures and sounds and registers *Core Game Components* and optionally *Additional Game Components*. Clients can be also configured to run one or more *A.I. Modules.*

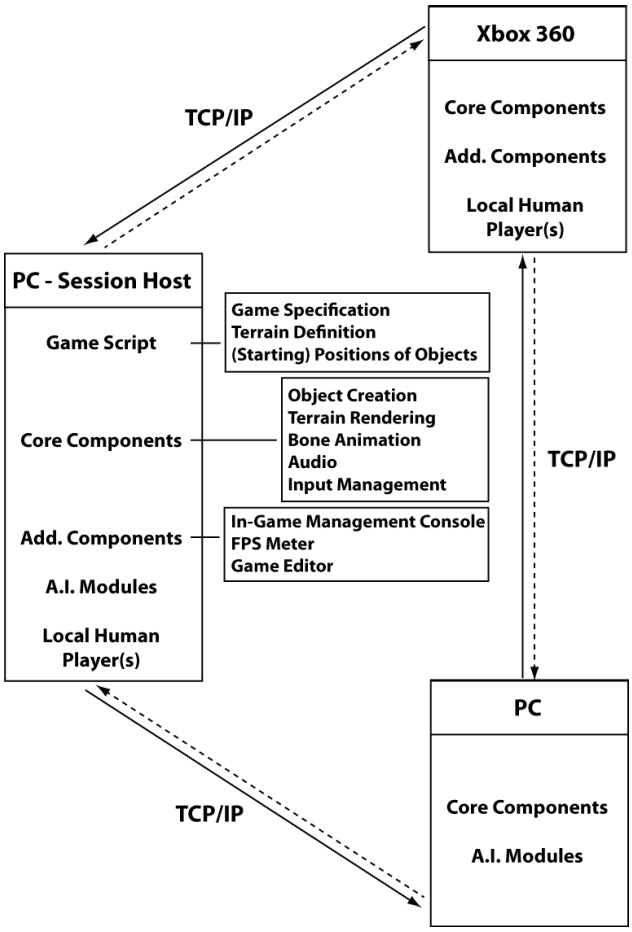

Figure 2. Example setup of a network game with three differently configured clients. One of clients is Session Host responsible for running Game Script with a definition of the game.

4.4. User-oriented architecture

MoonGate Engine was designed with different types of users in mind. Its three main parts represent three main categories of users that are targeted:

"Modders" (game modifications makers), level designers, people with no programming experience.

Modifications solely at *MoonGate Game Definition* level are suitable for people who are just getting to know the *MoonGate Engine* or real-time strategies in general. These users want on average to create a customized real-time strategy game as soon as possible without spending much time reading documentation or writing code. Changes at this level typically involve the creation of custom maps in map editor, replacing existing game assets with their own, tweaking properties of existing objects in configuration files and defining new objects out of existing components via configuration files. Changes at this level do not require programming skill.

Students, hobbyists, (individual) independent game developers, academics.

Users requiring a significantly customized game can do so just by editing existing ones and creating new *GameComponents* and *A.I. Modules* at *MoonGate Extended Engine* layer. This approach is recommended for advanced users with considerable programming skill. This way new objects can be created and new or modified types of behavior can be assigned to agents. Modifications at this level are also often used to test and visualize custom algorithms. Users making changes at this layer should be familiar with making changes at the *MoonGate Game Definition* layer**.**

Skilled individual independent developers, small teams of students or hobbyists.

By accessing the *MoonGate Core Engine* layer, engine users are have the option to control and modify every aspect of the game – be it visual representation, game logic or content pipeline with its extensions. This is necessary for using the engine outside the real-time strategy genre and for large modifications of application model and game logic as a whole.

This approach allows different types of users to use *MoonGate Engine* more quickly than traditional real-time strategy game engines, where one must understand their complete structure and have considerable programming skill in order to create a custom game.

5. Components

The creation of a flexible internal structure allows simple changes of any engine's part without the need of modification of other components. This approach greatly reduces the amount of possible errors and allows for the creation of specialized components that can be used without change in other projects in a plug-and-play style. Some of the most important components that provide functionality right out of the box are the following.

5.1. Terrain

The terrain component is one of the *MoonGate*'s more complex components. When considering what approach should be used to render terrain, quadtree-based terrain [15] was chosen mainly due to its simple and easily understandable implementation and sufficient performance. The implemented terrain supports bump mapping and texture blending.

Due to well defined interfaces that allow communication with other components, the default terrain component can be easily replaced with more sophisticated approaches that are able to handle rendering of a large and complex terrain with improved performance, as for example Geo-Mipmapping [16], without the need to change any other component or interface.

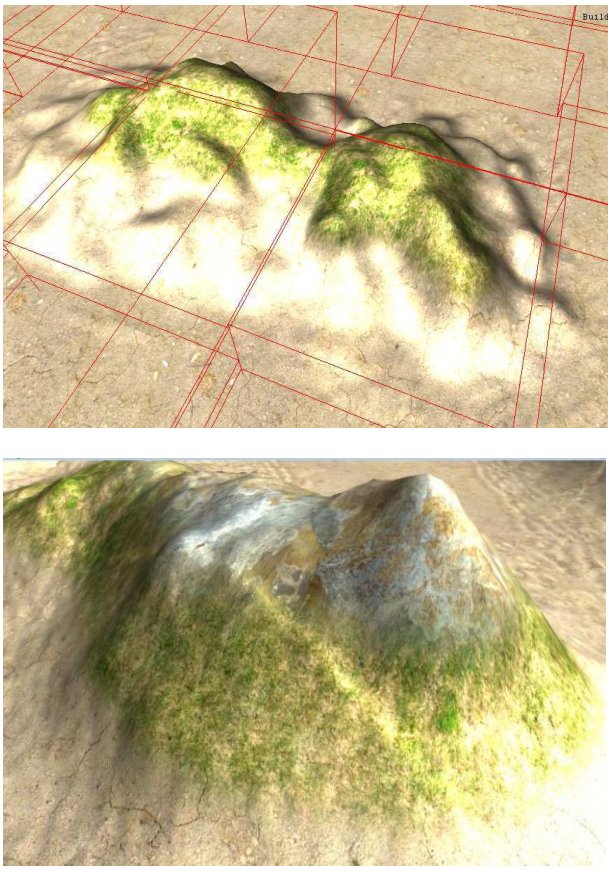

Figure 3. Terrain created by terrain component with visible quad-tree structure (up) and an example of terrain created by terrain component (bottom)

5.2. Objects creation

The heavy usage of flexible text-based formats like XML that are easily readable by humans and also computers ensures that not only various preferences, but also environment definition as a whole, can be modified and tweaked without the need of project recompilation.

One very important feature is also the definition of all static and dynamic objects like vegetation and units directly in XML files along with their representation and properties. *MoonGate Engine* goes beyond simple storing of independent values in XML files that can be commonly seen in most projects. A simple and intuitive hierarchy of definitions was designed and implemented in order to be used by *MoonGate*, in order to ensure fast learning and quick configuration.

```
<ModelInfo>
  <ID>Jeep</ID>
  <Path>models/units/jeep</Path>
  <Bones>
    <Bone>bone_turret</Bone>
  </Bones>
  <RenderMethod>ReplacementColor</RenderMethod>
</ModelInfo>

< Blueprint >
  <ID>LightAttackJeep</ID>
  <Behavior>StandardUnit</Behavior>
  <Model>Jeep</Model>
  <Category>Ground</Category>
  <Health>600</Health>
  <Armor>100</Armor>
  <TurnSpeed>0.3</ TurnSpeed >
  <MoveSpeed>0.9</ MoveSpeed >
  <LineOfSight>25</LineOfSight >
</ Blueprint >

<AnimationInfo>
    <Key>Jeep</Key>
    <Controllers>
      <Controller>
        <Bone>bone_turret</Bone>
        <ClassName>TurretControler</ClassName>
        <Attacks>GroundUnits</Attacks>
        <FireDist>80</FireDist>
        <ReloadTime>0.12</ReloadTime>
        <TurnSpeed>0.3</ TurnSpeed >
        <Bullet>
          <Type>SmallBullet</Type>
          <Sound>MachineGun</Sound>
          <MoveSpeed>130</MoveSpeed>
          <Damage>5</Damage>
          <Radius>3</Radius>
        </Bullet>
      </Controller>
    </Controllers>
</AnimationInfo>
```

Figure 4. Example of ModelInfo, Blueprint and AnimationInfo

In order to define an object we need to provide two XML definitions – *ModelInfo* and *Blueprint*. *ModelInfo* provides information about the 3D model used to visualize a given object (for example, .X file), object's bones that should be available for manipulation and a preferred rendering method. *Blueprint* describes an internal representation of the object and serves as a template for an engine's object factory. *Blueprint* also provides information about the object's behavior which is implemented as a library and thus is easily reusable. This way the user is able to describe a new type of object, whether static or dynamic, in just a few lines rather quickly and then immediately use it in the environment.

While the previous two definitions covered the creation of units, they did not provide a way to manipulate parts of the defined objects. In the genre of real-time strategies, units are often composed of several parts where each part acts autonomously at a certain level. A simplified example of this behavior can be a ship which consists of the main part – hull, several anti-ship cannons and several anti-aircraft cannons. These cannons are parts of the ship, but contain certain logic and animations which are characteristic for them. *AnimationInfo* describes individual parts of a model, their properties, animations and logic that controls them at the lowest level.

5.3. Particle systems framework

A particle component is a customizable lightweight 3D particle framework which uses point sprites. The particles are animated entirely on the graphics card using a custom vertex shader, so a large number of particles can be drawn with minimal CPU overhead.

Like the rest of the system, the particle component uses XML definitions to define properties of objects – in this case, particle systems. This approach allows for the creation and tweaking of the particle systems without the need to change the main game executable. There are several examples of particle systems in MoonGate that are available right out-of-the-box, for example, fire, smoke, explosions or trails.

5.4. Input management

The input management component provides a single point of access to user control input. Xbox 360 keyboard and Xbox 360 controller are fully supported on both platforms; on Windows platform it is also possible to use the mouse. Besides providing input, the input manager also provides the functionality of an in-game console. It is possible to define commands, execute them and display chosen values in real time in order to change *MoonGate*'s internal variables and state. The console provides also a

history of commands and an option to automatically complete commands. It is a useful tool not only during debugging, but also because it enables monitoring of the system's internal status.

5.5. Level editor

The level editor that comes with *MoonGate Engine* is not a standalone application like most editors that are available for real-time strategy games. *MoonGate* uses a custom built-in editor that is available directly from the game environment. Although this approach is more difficult to implement, it has a great advantage over the previously mentioned type – users are able to directly see the changes that they have made. There is no need to recompile the code or restart the application and the editor is available at any time.

MoonGate's editor works in three modes. The first is the "terrain painting" mode, where users "paint" different textures on the terrain. Painting is implemented as a change of weights of textures in the terrain's vertices. Every change is automatically saved, so the next time after the user starts the application, the most recent values are loaded. The second mode is the "static objects manipulation" mode. This mode permits manipulation with objects - users can add, remove, move, rotate and scale objects on the terrain, and immediately after the editor is closed, all changes are reflected into dynamic units' behavior, for example, the newly added objects that act as obstacles are avoided when units are moving.

The third mode is the "unit placement" mode in which users are able to add, remove and move dynamic objects – units which represent a complex unit with certain behavior controlled by the players. It is possible to create new units and assign them to various players or teams.

6. Performance

The use of XNA permits targeting besides the PC platform also Microsoft's Xbox 360 gaming console. Compared to other edutainment platforms focused on learning multiprocessing game development, graphics and media, as for example *Hydra Game Console* [17], which is based on Parallax Multiprocessing Propeller Chip with 32-bit RISC CPUs, Xbox 360 is by far superior in performance and also ease of use due to relying on C# language instead of using custom C-like and BASIC-like languages. This makes it a very interesting platform for students, academics and indie game developers.

In order to understand the sources of performance bottlenecks that occur mostly on the Xbox 360 platform, one must realize that the XNA Framework on Xbox 360 uses the compact version of Common Language Runtime (Compact CLR) to run a compiled intermediate language code. The Compact CLR is optimized for devices like PDAs and cell phones, where small size is more important than high performance. As such, the implementation of the Compact CLR on Xbox 360 does not optimize the floating-point code. Another problem is that the garbage collector on the Xbox 360 is not generational. As a consequence, when garbage collection occurs, all objects on the heap will be scanned to determine if they are alive or not. Finally, the just-in-time compiler does not optimize the code as well, so small routines are not inlined.

To overcome these difficulties, *MoonGate* uses various mechanisms like object pools [18], manual inlining of critical parts of the engine [19], along with combinations of design patterns (for example Prototype Factory). This is meant to avoid excessive garbage creation, keeping the heap small and simple in order to speed up the garbage collection, which, according to feedback from the XNA community, is currently one of the biggest performance bottlenecks. Due to the usage of both full and compact versions of CLR in XNA, *MoonGate*'s users are able to identify differences in performance on Xbox 360 and Windows platforms, learn by using code optimizations techniques and exploit multiprocessor environments on both platforms.

At the moment *MoonGate* runs at well over 60 FPS on most PC configurations and at more than 100 FPS on Xbox 360 with a medium-sized map and tens of objects, which is comparable to other free RTS engines.

7. User evaluation

In order to obtain feedback from *MoonGate Engine*'s users and to better know *MoonGate*'s audience, an online survey was conducted. Sixty-three participants – independent game developers, students, hobbyists and people experienced with real-time strategy games modifications who were using *MoonGate Engine* from several weeks to months took part in this survey.

The main goals of this survey were to identify the main ways in which *MoonGate Engine* is being used; identification of parts of the engine's architecture which are being most often customized, so future development could reflect these needs; and to better understand the community around *MoonGate Engine*.

Out of 63 participants, 13% were younger than 18 years old, 27% were between 18 and 25 years old, 38% were between 26 and 35 years old and 22% were older than 35 years.

The survey showed that *MoonGate* is used mostly by individuals - 84% of participants, while 12% of participants were using *MoonGate Engine* in teams with less than five members.

The results also showed that *MoonGate* is currently being used primarily as a base for an independently developed real-time strategy game; a test bed for A.I. algorithms; as an advanced XNA tutorial; and as a test bed for HLSL shaders. For complete results related to this question, see Figure 5.

Users wishing to modify *MoonGate Engine* in order to create custom a game prefer making changes at *Game Definition* and *Core Engine* layers. Almost half of the participants preferred changes at these layers to just changing *Game Definition* configuration files and importing custom assets, and making changes to all three layers (Figure 6).

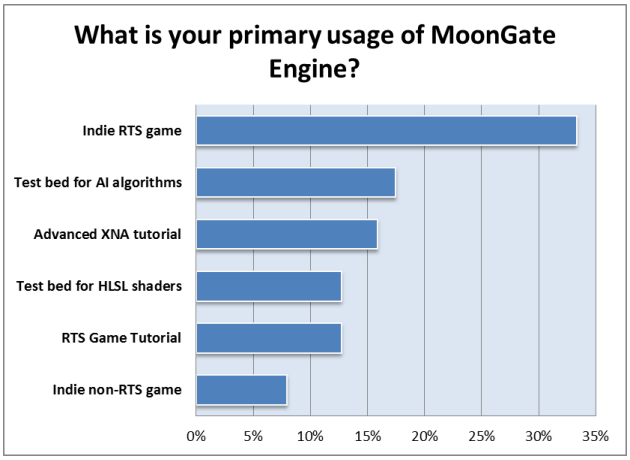

Figure 5. Survey results showed that *MoonGate Engine* is currently being used primarily as a base for independently developed real-time strategy games and as a test bed for A.I. algorithms

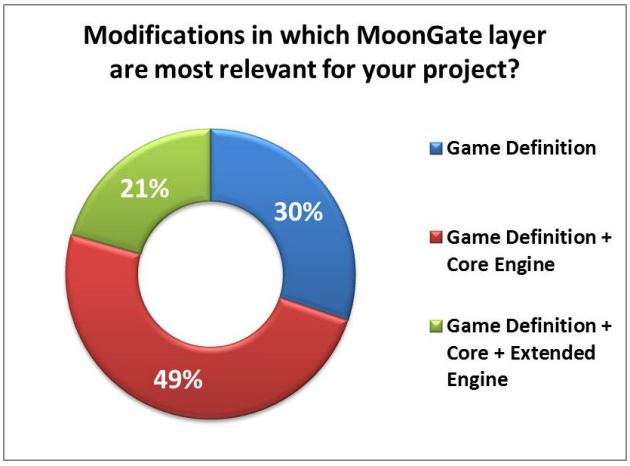

Figure 6. Users wishing to modify *MoonGate Engine* prefer making changes at Game Definition and Core Engine layers to just changing Game Definition files or making changes to all three layers

For future development, participants prefer the creation of a reusable set of GUI components – 19%; in contrast to improvements in material manager – 15%; and improvements of *MoonGate*'s networking abilities – 14%.

The survey also showed that a quarter of the participants had previous experience with other real-time strategy engines.

Survey results have confirmed an hypothesis that individuals and small teams of hobbyists can benefit from a multiplatform open source real-time strategy engine in various ways – be it development of an independent game or A.I. algorithms testing and that engine's component-based structure that allows quicker development by providing easy access to those layers of the engine in which change is relevant for the user.

8. Conclusion

In this paper, *MoonGate* – an open source, real-time strategy game engine for Xbox 360 and Windows platforms was presented as a tool of choice, not only for indie game developers, but also for students and hobbyists. *MoonGate*'s main goal is to attract them to learn by creating highly interactive and visually attractive games and give them an opportunity to easily study various problems by providing an easily understandable and highly customizable educational environment which incorporates the best features of modern commercial games, visualization environments and game middleware. What is important, MoonGate is the first open source starter kit and real-time strategy engine for Xbox 360.

MoonGate promotes simple extensibility through usage of reusable components, custom content pipeline extensions that permits easy creation and import of game content and tweaking without the need of source code recompilation. Its flexibility not only allows for the creation of a game in one of the most popular game genres, but it also allows for the exploitation of Xbox 360's hardware for a variety of tasks, ranging from shader development, rendering of terrain through particle systems to visualization of AI algorithms, which are very tightly bound to this game genre.

At the moment *MoonGate* is used by individuals and small teams that are using it as a base for custom strategy games and as a platform for experimenting with algorithms from various research areas.

In the near future, the network component that is still in the beta version will be replaced with a more robust version that will be able to handle synchronization among connected users with better results. Since the XNA Framework does not provide standard graphics user interface controls out of the box, a component that allows for the usage of these controls will be added to *MoonGate*.

References

[1] R. Butt and S. J. Johansson, 2009. Where do we go now?: anytime algorithms for path planning. *In Proceedings of the 4th International Conference on Foundations of Digital Games*, pp. 248-255.

[2] M. Buro and J. Bergsma and D. Deutscher and T. Furtak and F. Sailer and D. Tom and N. Wiebe, 2006. *AI Systems Designs for the First RTS-Game AI Competition*. [Online]. Available: http://www.cs.ualberta.ca/~mburo/ps/ortscomp06.pdf [Accessed: Feb. 10, 2011].

[3] M. Buro, 2003. Real-Time Strategy Games: A New AI Research Challenge. *In International Joint Conferences on Artificial Intelligence*, pp. 1534-1535.

[4] M. Sharma and M. Holmes, 2007. Transfer Learning in Real-Time Strategy Games Using Hybrid CBR/RL. *In Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pp. 1041-1046.

[5] J. Orkin, 2003. Applying Goal-Oriented Action Planning to Games. *In AI Game Programming Wisdom II*, Charles River Media.

[6] M. Buro, 2005. *Call for AI Research in RTS Games*. [Online].Available: http://www.cs.ualberta.ca/~mburo/ps/RTS-AAAI04.pdf [Accessed: Feb. 8, 2010].

[7] The MoonGate Engine [Online]. Available: http://mgengine.blogspot.com/ [Accessed: Feb. 15, 2011].

[8] XNA Facts 'N Stats [Online]. Available: http://creators.xna.com/assets/cms/docs/marketing/GDC09/XNA%20Facts%20N%20Stats.docx [Accessed: Jan. 22, 2011].

[9] ORTS – A Free Software RTS Game Engine [Online]. Available:http://www.cs.ualberta.ca/~mburo/orts/ [Accessed: Jan. 25, 2011].

[10] The Spring Project [Online]. Available: http://springrts.com/ [Accessed: Feb. 15, 2011].

[11] Glest – The Free Real-Time Strategy Game [Online]. Available: http://glest.org/en/index.php [Accessed: Feb. 15, 2011].

[12] Educomp Solutions [Online]. Available: http://www.educomp.com/ [Accessed: Feb. 15, 2011].

[13] K. D. Forbus, J. V. Mahoney and K. Dill, 2002. How qualitative spatial reasoning can improve strategy game AIs. *In IEEE Intelligent Systems*.

[14] Neverwinter Nights [Online]. Available: http://nwn.bioware.com/ [Accessed: Feb. 14, 2011].

[15] R. Pajarola, Overview of Quadtree-based Terrain Triangulation and Visualization [Online]. Available: http://vmml.ifi.uzh.ch/files/pdf/publications/UCI-ICS-02-01.pdf [Accessed: Feb. 15, 2011].

[16] W.H. de Boer, Fast Terrain Rendering Using Geometrical MipMapping [Online]. Available: http://www.flipcode.com/archives/article_geomipmaps.pdf [Accessed: Feb. 15, 2011].

[17] Hydra Game Development Kit [Online]. Available: http://www.xgamestation.com/ [Accessed: Jan. 22, 2011].

[18] Improving Performance with Object Pooling [Online]. Available: http://msdn.microsoft.com/en-us/library/ms682822(v=vs.85).aspx [Accessed: Feb. 16, 2011].

[19] GDC 2008: Understanding XNA Framework Performance [Online]. Available: http://www.microsoft.com/downloads/en/details.aspx? FamilyID=b11ad912-4158-44cc-a771-a5e044f7e3bb&displaylang=en [Accessed: Feb. 16, 2011].

Author Biographies

Rudolf Kajan is a PhD student at Faculty of Information Technology, Brno University of Technology, Czech Republic. He is a member of Graph@FIT research group. His research interests include computer graphics and processing multimedia data in heterogeneous distributed environments.

Adam Herout received his PhD from Faculty of Information Technology, Brno University of Technology, Czech Republic, where he works as an associate professor and leads the Graph@FIT research group. His research interests include fast algorithms and hardware acceleration in computer vision and graphics.