Guff: A Game Development Tool

Luis Valente, Aura Conci (Advisor)
Universidade Federal Fluminense – Instituto de Computação
Rua Passo da Pátria, 156 Bloco E Sala 452 - 24.210-240 – Niterói – RJ
{lvalente,aconci}@ic.uff.br

Abstract

Due to the real time nature of computer games, the main concerns of game developers have been related to efficiency of algorithm execution and visual presentation, instead of code reusability and maintenance. In order to support these concerns, game developers usually have been implementing all the required functionality to build a game, from scratch, in new projects.

However, the complexity in game projects has been increasing over time, rendering such practices infeasible. Then, it is necessary to seek new approaches for game development.

This work describes a game development tool (developed for a Master's Thesis), which applies an approach that can address this issue: a framework. The Guff framework is an easy to use tool that features automatic resource management on behalf of the developer, a state machine approach to specify and manage game levels, and extensive employment of free and open source libraries to avoid implementing already available functionalities.

1. Motivation

Older computer games (like the ones from the beginning of the 80's) were very simple due to few resources available at that time, more precisely, low processing power and little amount of memory. Therefore, the main concerns of game developers were related to efficient algorithm execution and generated code size. The code was to be very efficient or the machine would not be able to run it in an acceptable way. If the code were not compact, it would not fit in the machine's memory. To fulfill these requirements, the developers usually applied assembly language. A common approach was to eliminate intermediate layers (between application and hardware), in order to access the hardware directly, so as to obtain the best performance possible.

Indeed, this necessity has become a common practice in game development. Due to the "maximum efficiency" criteria, the decision to implement every feature required to build a game, in each new project, has become a common behavior. Practices like this are known as "not built here" [1], which means "if we did not built it, then it must be a bad choice". However, as the complexity in game projects increases, such practice is becoming infeasible.

The utilization of assembly language in computer games is becoming less of an issue as computer processing power grows over time. In 1993, a groundbreaking game called Doom [2] was released. Doom is one of the most seminal games in the first-person shooter [3] genre. It was implemented almost entirely with the C programming language. The assembly language was applied in a few parts of this project. This game, somewhat, helped to reduce the opposition against using higher level languages in game development.

In 1994, the structure of a typical game from that time (for instance, a side-scroller game [3]) would be comparable to the one depicted in figure 1.

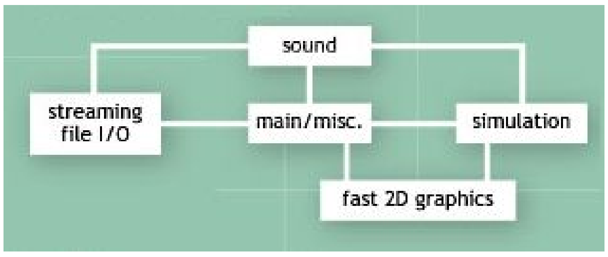


Figure 1. 2D game from 1994 [4]

The nodes in figure 1 correspond to application modules. The connections between the modules represent dependencies among them. The main module acts a controller to the other modules, and aggregates functionalities which are not complex enough to form a new module.

Figure 2 (at the end of this work) illustrates a structure of a game from 1996. In that period, games which employed unrestricted three-dimensional visualization, like Quake [5], were released. The first 3D accelerated graphics boards arose during that time, which can be considered as the beginning of the 3D engine era. In general terms, a 3D engine is an application that implements an architecture for computer games that employ three-dimensional visualization. By observing figure 2, it is possible to notice a slight increase in the complexity of a computer game structure.

Today, a computer game structure is more complex. For instance, figure 3 (at the end of this work) depicts a possible structure for a single-player game from 2004.

By comparing figures 1, 2, and 3, it should be easier to understand why practices such as “not built here” are becoming infeasible nowadays. Current game projects last from 24 to 36 months, typically [1]. Approaches which help to diminish that time span (and cost, consequently) are, without question, helpful.

This work describes a game development tool that is based on the framework concept, the Guff framework. This tool was developed for a Master's Thesis [3].

The Guff framework approach to game development features automatic management of all available resources on behalf of the developer, use of free and open source libraries in order to avoid functionality duplication, and game levels specification and management through a state machine.

2. The Guff Framework (games-uff)

Frameworks are a set of classes which form a reusable architecture for a family of systems [6]. The Guff framework [3] was implemented with the C++ programming language and applies paradigms like object-oriented programming, design patterns [6] and automatic resource management [7]. This tool defines a reusable architecture for games, and initiates a graduate research field at Universidade Federal Fluminense on game development. Guff comprises two main parts: the application layer and the toolkit. The application layer defines the interfaces and program architecture. The toolkit stands for a set of auxiliary classes which developers may use to build new applications.

In general terms, these are the requirements for Guff:

- Offer an easy to use environment for game development;
- Offer a possible architecture for games, by providing an interface to model the game levels as states;
- Manage all resources automatically on behalf of the developer;
- Offer an auxiliary tool set to help in application development;
- Work in several operating systems (portability);
- Apply free and renowned libraries whenever possible, to avoid reimplementation of functionality already available.

Table 1. Libraries applied in Guff implementation

Library	Location
OpenGL	http://www.opengl.org
SDL	http://www.libsdl.org
GLEW	http://glew.sourceforge.net
boost	http://www.boost.org
lib3ds	http://lib3ds.sourceforge.net
audiere	http://audiere.sourceforge.net
FTGL	http://homepages.paradise.net.nz/henryj/code
Lua	http://www.lua.org
DevIL	http://www.imagelib.org

Table 1 lists the libraries which Guff employs in its design. These libraries have implementations for many operating systems. Another objective of applying these libraries is to fulfill the portability requirement.

OpenGL is a three-dimensional graphics library for real time visualization and modeling. Its development was started by Silicon Graphics in 1992, and today it remains as a Computer Graphics industry standard.

The SDL (Simple DirectMedia Layer) is an open source multimedia library that abstracts subsystems such as input devices, audio, and windowing systems, among others.

The GLEW (OpenGL Extension Wrangler) open source library was designed to automate the initialization of OpenGL extensions available in the graphics board. These extensions provide new functionalities beyond the current OpenGL version. Its main developers are M. Ikits and M. Magallon.

The boost project was started by members of the C++ Standard Committee. It provides a wide range of libraries related to concurrency, generic programming, Math, and memory management, among others. One of its main goals is to establish best practices for library implementations, in order to make them suitable for standardization.

The lib3ds library handles loading and parsing of the 3ds file format, which was originally designed for the animation and modeling software 3ds max. This library is an open source project started by J. E. Hoffman.

The audiere library was designed to perform input/output operations, decoding, and mixing of audio data. Its development was started by C. Austin.

FTGL was developed by H. Maddocks in order to provide font loading and rendering for OpenGL applications.

Lua is a scripting language designed to extend applications. It can used as a general purpose language (stand-alone) or as an embedded application module. It was designed and implemented by R. Ierusalimschy, W. Celes, and L. H. de Figueiredo at Tecgraf (PUC-Rio).

The DevIL (Developer's Image Library), which was originally developed by D. Woods, is an open source image library that is able to handle several types of image formats.

2.1. Application Layer

The application layer is modeled as a state machine. The motivation to adopt this model is that a game can be organized into a set of states. For example, a hypothetical game might comprise these stages: an introduction, a main menu, the main game, and a special menu that is accessed in-game. Figure 4 illustrates this state machine.

The purpose of this model is to have a state corresponding to each one of those stages, in order to ease their specification and management.

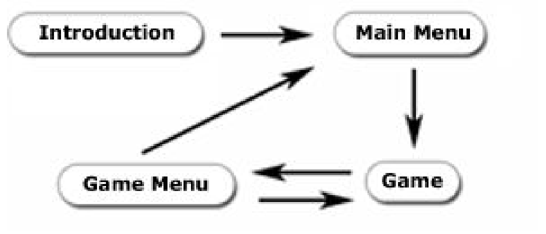


Figure 4. Hypothetical game state machine

Guff defines a class hierarchy that makes possible the definition of single states and state groups. The state groups are a collection of states. Figure 5 illustrates this class hierarchy.

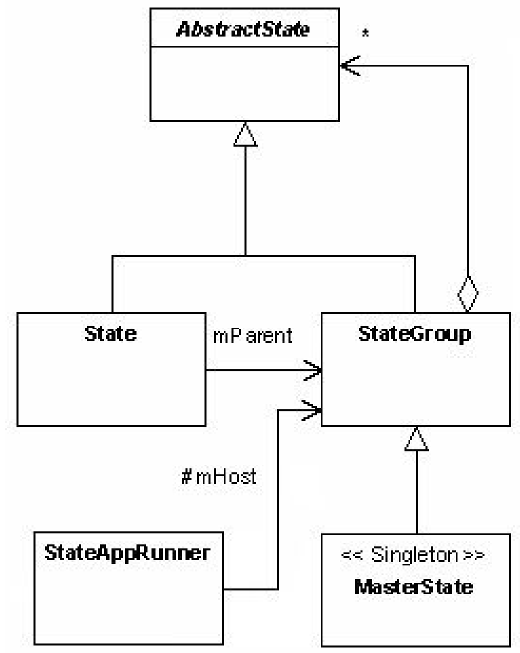


Figure 5. The Guff state machine class diagram

The state classes feature methods which correspond to application events. Those events can be classified in the following categories: system events, events related to states, and events related to the game execution model.

The system events include the windowing system events (such as window activation, window deactivation and application window resize), and user input events (mouse button press and release, mouse move, and keyboard press and release).

The events related to states occur on state change operations, such as state enter and state exit.

A game execution model is an approach to organize the execution of the various tasks in a game. These tasks can be classified in the following categories: input data acquisition, processing and presentation of results [8]. The input system corresponds to management of input devices, such as keyboard, mouse, and joystick. The processing stage executes update tasks that determine the current game state, such as the animation interpolation, the physics simulation, the game AI (Artificial Intelligence), and the game logic *(e.g.* the application of the game rules). The presentation stage demonstrates the results from other stages, using audio and video.

In Guff, the `StateAppRunner` class is responsible for implementing an execution model, and delegating event handling to the application (*e.g.* its states). Figure 6 illustrates the Guff game execution model.

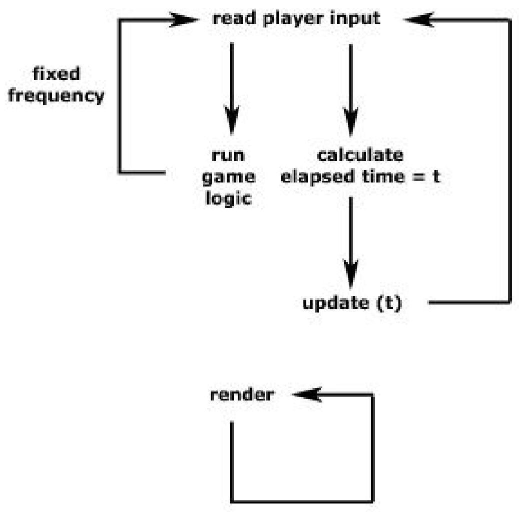


Figure 6. The Guff game execution model

There are three distinct cycles in this model: the rendering cycle, the update cycle, and the game logic execution cycle. The update cycle and the game logic cycle belong to the processing stage. The model implementation is single-threaded. The main purpose is to avoid issues related to concurrency (such as explicit synchronization to data access).

The game logic execution cycle runs at a fixed frequency. The motivation to this approach is that some tasks may not present significant results if they are executed repeatedly in a very brief time interval (such as the game AI and application of game rules). The update cycle can be used to run tasks which may present better results if they are executed more frequently, such as animation interpolation.

Guff automatically performs associations among parent and children states. All kinds of states (single or group) have a parent. The `MasterState` class is the default parent for all states in Guff. This state is set as a parent state if the developer does not specify any. As an example, this code listing defines the hierarchy illustrated in figure 4:

```
class Introduction : public State {
public:
  Introduction ()
   : State ("introduction") {}
};

class MainMenu : public State {
public:
  MainMenu ()
   : State ("main menu") {}
};

class GameMenu : public State {
public:
  GameMenu ()
   : State ("game menu") {}
};

class Game : public State {
public:
  Game ()
   : State ("game") {}
};
```

To create this hierarchy, it is enough to declare an instance of each state class. In this example, the root of the hierarchy is the master state, which has four children. Each class constructor defines the state name. This is an important property, which is used in state change operations.

The state change operations are invoked by the states themselves. The request is handed to their parents, which are responsible for effectively executing the operation. There are two types of state change operations: temporary changes and definitive changes. These operations accept the name of the state which should be changed to, as a parameter. For example, the following code snippet request a definitive state change from "introduction" to "main menu":

```
void Introduction::OnUpdate (float time)
{
   mParent->changeState ("main menu");
}
```

The temporary state change operation can be used when it is required to remember the current state prior to the operation invocation. Referring to figure 4 as an example, this may happen if it were necessary to change to the "game menu", and restore whatever state was current before the operation. The definitive state change operation can be used when this behavior is not required. The next code listing exemplifies temporary state change operations:

```
void Game::OnUpdate (float time)
{
   command = getInputCommands ();

   if (command == SHOW_GAME_MENU)
    {
      mParent->pushState ("game menu");
      return;
    }

   ...

}

void GameMenu::OnUpdate (float time)
{

   command = getInputCommands ();

   if (command == EXIT_GAME_MENU)
    {
      mParent->popState ();
      return;
    }

   ...
}
```

2.2. The Toolkit

The Guff toolkit applies the automatic resource management paradigm [7] for all of its services, which provide functionalities related to visualization, automatic management of third party libraries, application configuration, input devices, Math, audio, and utilities. As a consequence, this fact contributes to turn Guff into an easy to use tool. The available functionalities are summarized as follows:

- Texture creation and management, using bmp, jpg, and tga, among other file formats;
- Three-dimensional model loading and management (3ds files);
- True type font management;
- Streaming audio (mp3, ogg, and wav, among other file formats);
- Automatic OpenGL extension loading, so as to take advantage of new features available in the graphics board;
- Rendering of Quake III [9] scenes;
- Several types of cameras are available;
- Lua programming language integration, which Guff uses to define application parameters through script files.

Figure 7 depicts a relationship among the available modules and their respective namespaces.

Some libraries applied in the Guff design must be initialized before utilization and finalized when the application is over. In order to make this process robust and reliable, the initialization module applies the automatic resource management paradigm to solve this issue. This module applies this technique for SDL, GLEW, and DevIL libraries, for example.

There are two interesting advantages derived from this approach. Firstly, it is possible to initialize and finalize libraries correctly, even if exceptions (fatal errors) occur during the application life cycle. The second advantage is the possibility to handle library dependencies. For example, if library A requires that library B is initialized first, it suffices to encapsulate both libraries as automatic objects, and declare them in the required order. An automatic object is a class which is responsible for releasing a resource to the system. Thus, the finalization procedure (which should happen in the reverse order) will be performed correctly.

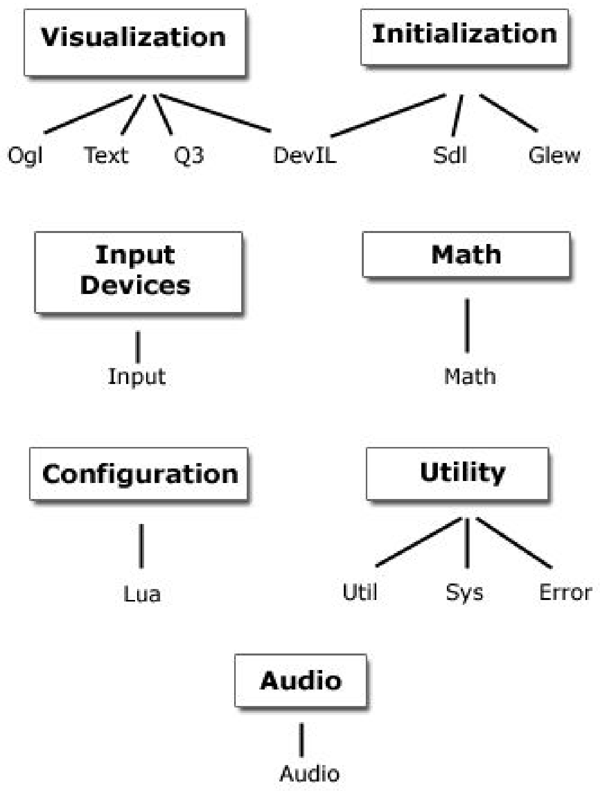


Figure 7. Guff modules and their namespaces

The configuration module defines many parameters which affect the behavior of Guff applications. This module applies the Lua language in its implementation. The developer can define these parameters through the config.lua script, which is mandatory for all applications. For example, such script holds values for the video configuration (OpenGL). This approach is interesting, because it makes possible to define many parameters without recompiling the application.

The input devices module offers classes to encapsulate the keyboard and mouse devices. These classes have methods to query the current device state, such as position (mouse) and key states (mouse and keyboard).

The Math module defines classes which represent mathematical entities such as vectors, planes, matrices, quaternions, frusta, and bounding boxes. A bounding box is the smallest volume which encompasses some geometry. The Guff toolkit represents bounding boxes as AABBs (Axis Aligned Bounding Boxes). An AABB is a box aligned to the global X, Y, and Z axes.

In order to help in visibility determination operations, the toolkit makes available some intersection tests against the frustum (viewing volume), such as sphere versus frustum, bounding box versus frustum, and cube versus frustum.

The utility module provides tools like clocks, stopwatches, error handling functions, and a base class for implementing resource caches. This base class represents the foundation for other caches in the toolkit.

The audio module provides loading and playing of mp3, ogg, and wav files, among other file formats. The developer may opt to use streaming, if desirable. The streaming operation means the application will load the sound data on demand. In other words, it will not load the file entirely to memory, once. This feature is helpful in playing large audio files.

This module implements an audio manager, which any application should use to load audio files. The main responsibility of the audio manager is avoid loading a sound file more than once. To fulfill this requirement, this manager implements a cache. When the application needs to load an audio file, it sends a request to the cache in order to load that file. Then, the audio manager queries its internal cache to check if the file has already been loaded. If it is so, it returns a reference to the audio stream. If the cache does not have a reference to the audio stream, it loads the corresponding file, adds an entry to the cache and returns a reference to the new audio stream. The class which represents an audio stream applies the automatic resource management paradigm. Hence, it is not required to release this resource explicitly. As an interesting consequence, the audio manager does not need to manage this resource altogether.

The visualization module implementation employs OpenGL as its basis API. This module defines three managers: a texture manager, a font manager, and a 3D model manager. Similarly to the audio manager, the main responsibility of these classes is to avoid resource duplication in memory.

The texture manager reads image files from disk and uses the data to create textures. This class handles many image formats such as bmp, jpg, tga, and png, among others. This manager file loading operations may also generate mipmaps automatically. The class which represents the texture itself also applies the automatic resource management paradigm. This class implementation embodies a reference counter, in order to provide a simple way to share resources.

The 3D model manager supports static meshes using the 3ds file format. The only restriction regarding 3D models, is that each polygon mesh which belongs to them must apply only one material. The purpose of such restriction is to minimize state change during rendering, and to send each mesh in its entirety to the graphics board, as one geometry batch. State change minimization is important to avoid reconfiguration of the graphics pipeline during rendering, which can be a costly operation [10].

The font manager can be used to load true type fonts from disk, in order to perform text rendering operations. The font rendering API provides some options regarding character rendering, such as: texture mapping (a polygon with texture mapping represents each character), bitmaps (a binary image represents each character), lines (lines represent the character silhouette), pixmaps (a non-binary image represents each character), and polygons (a polygon mesh represents each character). In addition, the toolkit implements functions which help the developer to establish text rendering projections, in such a way that the text is always on top of the 3D scene.

The Guff also provides classes related to point of view specification, which correspond to cameras, orthographic projections and perspective projections. There are three types of cameras available in Guff: free cameras, orbit cameras, and tracking cameras.

The camera implementations use the matrix classes defined in the Math module, in order to avoid the gimbal lock. The gimbal lock is a phenomenon inherent to Euler angle representation, when applied to orientation specification. To specify an orientation with Euler angles, one must apply sequential rotations in the X, Y, and Z axes. The gimbal lock can manifest itself when an angle of $\pm 90^{\circ}$ for the second rotation, causes the first and third angles to rotate about the same axis (they should be independent) [11].

The free cameras have six degrees of freedom (three for translation and three for rotation). The orbit cameras help to specify a point of view which spins around a reference point. The tracking cameras permit a reference point specification so that the camera will always be oriented towards it. The positioning of tracking cameras is arbitrary. The orientation is calculated in such a way that the camera is always "upside" when compared to the tracked object.

Regarding rendering of complex scenes, the Guff toolkit has classes that handle Quake III scenes (often referred to as map files). Quake III is a first-person shooter [3] game, which presents indoor environments with reasonable detailed geometry. The scene geometry may be modeled with polygon meshes, billboards, and biquadratic Bèzier surfaces. It is relatively easy to find new maps and map editing tools related to this game, so this remains as the main motivation to use them.

The Quake III map implementation employs a BSP (Binary Space Partition) tree, which holds all scene geometry. This data structure is adequate to represent indoor scenes, which are prevailing in first-person shooter games such as Quake III.

Whenever is necessary to render the BSP tree geometry, the scene renderer queries it to find out the current player position in the scene. Then, it applies frustum culling and queries the PVS in order to perform visibility determination. The PVS (Potentially Visible Set) is a data structure which holds information about which rooms are visible from some other room. The PVS calculation is a costly process. Therefore, its contents are calculated offline.

The Guff toolkit also presents classes which perform collision detection of rays and spheres against the BSP tree. These operations are helpful to avoid having the user cross through the scene.

3. Experimental Results

In order to evaluate the performance of Guff applications, we implemented a test program that simulates a virtual tour in a Quake III map and executes a playlist of background musics. The following list summarizes the test environment:

- Intel Pentium 3 600E processor;
- GeForce 4 MX graphics board with 64 megabytes of memory, NVidia video diver version 56.72, AGP 2X;
- 256 megabytes of system memory;
- Windows XP Professional;
- Creative Labs SoundBlaster Live! audio board.

The test Quake III map is the q3dm1 scene file. This file ships with the official Quake III demo. Table 2 lists the statistics of this file.

Table 2. Test scene statistics

Type	Amount
Vertices	13978
Textures	94
Lightmaps	9
Scene polygon faces	1942
Faces of 3D models in the scene	42
Faces generated for the parametric surfaces	113

The parameter applied to evaluate performance was the amount of frames per second generated by the application. A frame represents a complete cycle of application execution that generates a image on the screen.

All the tests were run in a 800x600 window, with 32 bits of color depth, and a 24 bit z-buffer. Table 3 lists the results.

Table 3. Test results

Settings	Best frame rate	Worst frame rate	Mean frame rate
Fullscreen off Vsync off	135	57	84.7
Fullscreen off Vsync on	61	31	55.8
Fullscreen on Vsync off	242	76	125.8
Fullscreen on Vsync on	61	42	56.2

The Vsync parameter tells the application to synchronize its execution with the monitor vertical synchronization rate.

Figures 8 and 9 illustrates the test scene file rendered by Guff and by Quake III, respectively. It is important to notice that it is not possible to compare directly the test application and Quake III, as this game performs many operations (such as the game Artificial Intelligence, network processing, and advanced rendering effects, for example) not implemented in the test application.

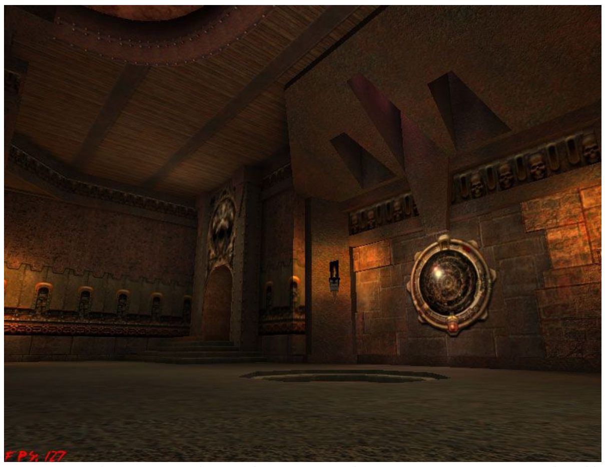


Figure 8. The q3dm1 scene file rendered by Guff

As an example of easy usage (and automatic resource management), the following code listing applies:

```
void foo ()
{
  // t1 loads the default texture, a
  // checkerboard pattern
  Texture2 t1;

  // t2 loads the abc.jpg texture
  Texture2 t2 ("abc.jpg");
  // t2 releases the resources allocated for
  // abc.jpg, and now references the default
  // texture
  t2 = t1;

  // t1 loads the xyz.tga texture, and t2
  // still references the default texture
  t1.load ("xyz.tga");
}

// the resources allocated for xyz.tga and for
// the default texture are released
// automatically when the execution flow exits
// the function
```

4. Conclusions

Computer games apply concepts from several Computer Science fields such as Computer Graphics, Software Engineering, Artificial Intelligence, and Computer Networks, among others.

Game developers usually try to exploit at most the computational resources available to them, in order to maintain interactivity. Therefore, they frequently implement all functionality required to build a game, in each project. This practice is is becoming infeasible as the complexity in such projects increases over time.

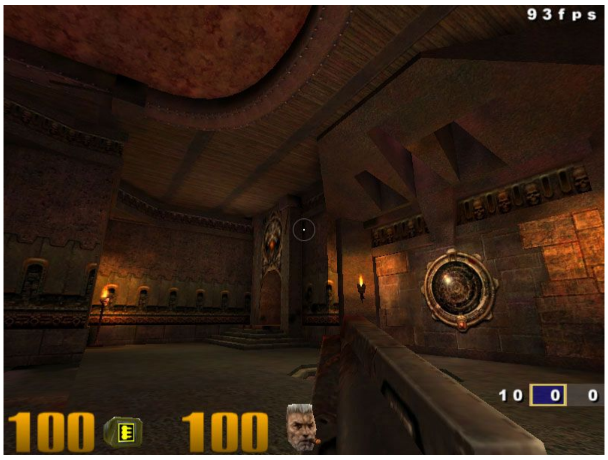


Figure 9. The q3dm1 scene file rendered by Quake III

Guff is a game framework that proposes a reusable game architecture for games. This type of tool is an alternative to combat the "not built here" syndrome. This tool has to main parts: the application layer and the toolkit. The application layer describes a program architecture using a state machine approach. The state machine controls a set of states that represents game levels. The toolkit represents a set of classes that can be used to develop game applications.

Frameworks are hard to design because it is necessary to know in advance what the developer is going to need. Designing generic game frameworks is even harder, because some techniques applied to game development are contradictory. For example, games which present indoor scenes usually apply algorithms that are not adequate to process outdoor scenes, and vice versa. In spite of these issues, it is very important to develop such tools in order to diminish the impact of the increasing complexity in game development.

Future works related to Guff would be extending its core functionality by adding new modules (for example, networking) and refining modules already present (for instance, incorporating a skeleton animation feature in three-dimensional models). Another interesting approach is to design a plugin system. Such system would allow to define conflicting functionality independently (for example, outdoor and indoor scene rendering), and to select the most appropriate one at runtime, thus providing great flexibility.

Finally, the following aspects proposed in Guff should be highlighted:

• Extensive application of free and open source libraries, to avoid implementing already available functionality. This approach helps to fulfill the portability requirement, and is the basis of the auxiliary Guff toolkit;

• The automatic resource management paradigm is a fundamental concept in the Guff design. Its employment is a simple way to minimize errors related to resources, raise application robustness and reliability. Consequently, this concept contributes to make Guff an easy to use tool.

• The state machine from the application layer facilitates the specification and management of game levels. This approach accomplishes an architecture for game development.

Acknowledgments. The authors thank the financial support of CAPES. The second author also thanks CNPq (Grant PQ No. 302436/2004-9).

References

[1] Rollings, A., and D. Morris, *Game Architecture and Design: A New Edition*, New Riders, Indianapolis, 2003.

[2] Wikipedia.org, "Doom – Wikipedia, the free encyclopedia", Available online at http://en.wikipedia.org/wiki/Doom (04/25/2005).

[3] Valente, L., *Guff: Um Sistema para Desenvolvimento de Jogos*, Master's Thesis, Universidade Federal Fluminense, Niterói, 2005 [in Portuguese].

[4] J. Blow, “Game development: Harder Than You Think”, *ACM Queue*, ACM, New York, 2004, pp. 29-37.

[5] id Software. "Quake". Available online at http://www.idsoftware.com/games/quake/quake (04/25/2005)

[6] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Massachusetts, 1995.

[7] L. Valente, and A. Conci, “Automatic Resource Management as a C++ Design Pattern”, *Simpósio Brasileiro de Jogos para Computador e Entretenimento Digital,* Curitiba, 2004, CD-ROM.

[8] Dalmau, D., *Core Techniques and Algorithms in Game Programming*, New Riders, Indianapolis, 2003.

[9] id Software, "Quake III Arena". Available online at http://www.idsoftware.com/games/quake/quake3-arena (04/25/2005)

[10] Valente, L., *Representação de Cenas Tridimensionais: Grafo de Cenas*, Research report RT-03/04, Universidade Federal Fluminense, Niterói, 2004 [in Portuguese].

[11] Parberry, I., and F. Dunn, *3D Math Primer for Graphics and Game Development*, Wordware, Plano, 2002.

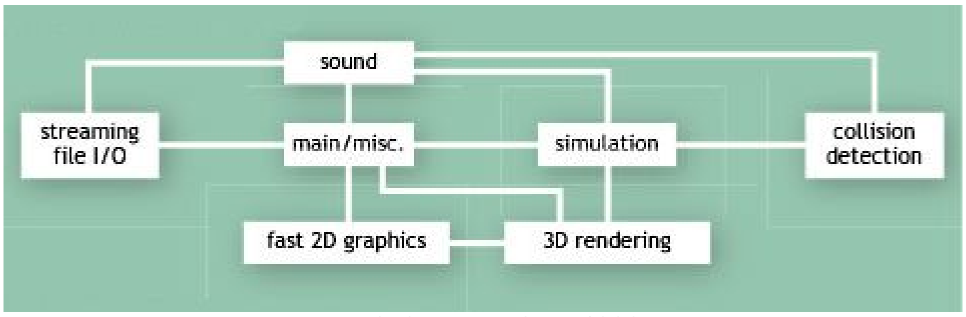
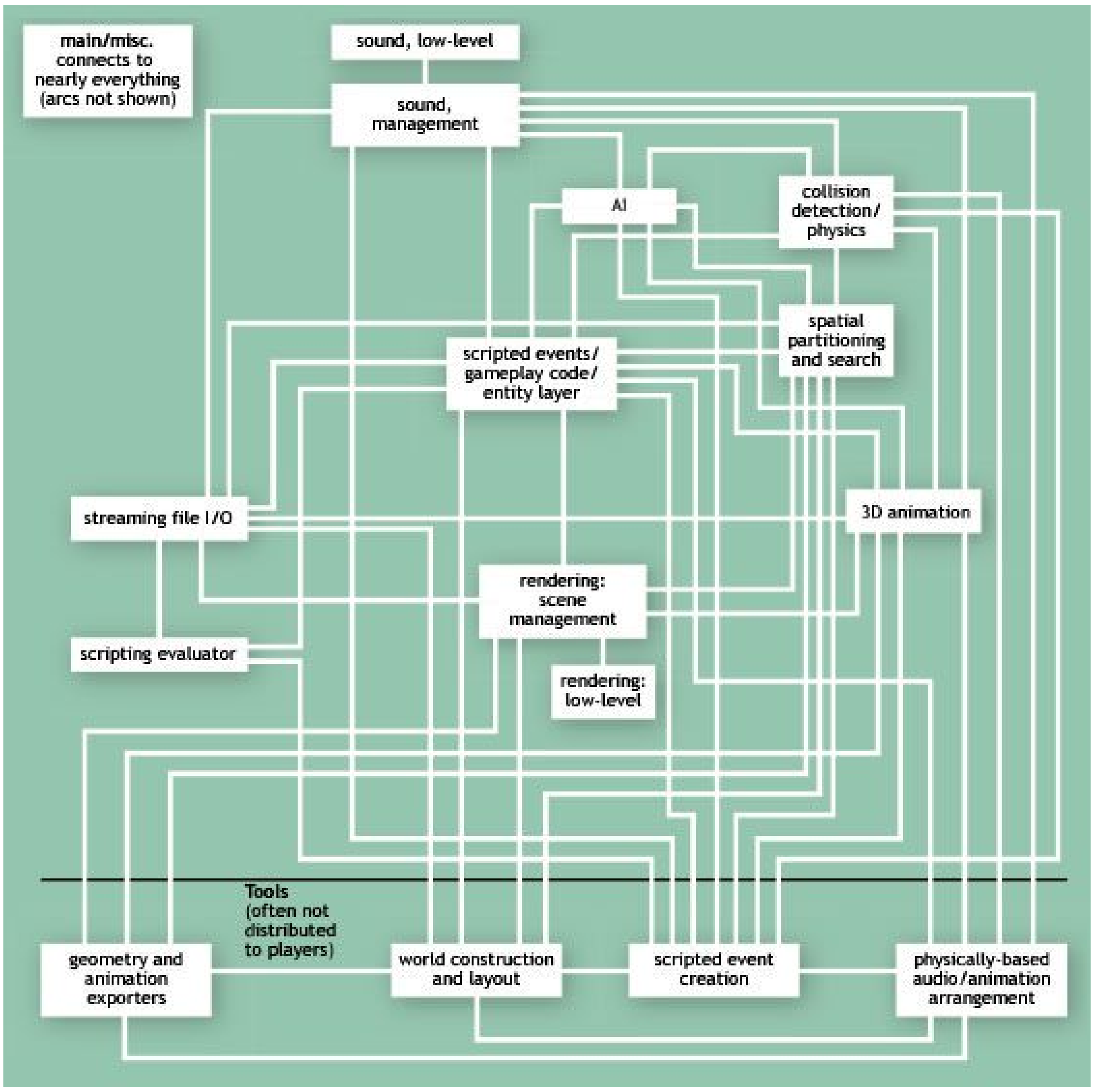


Figure 2. 3D game from 1996 [4]


main/misc. connects to nearly everything (arcs not shown)
sound, low-level
sound, management
AI
collision detection/ physics
spatial partitioning and search
scripted events/ gameplay code/ entity layer
streaming file I/O
3D animation
rendering: scene management
scripting evaluator
rendering: low-level
Tools (often not distributed to players)
geometry and animation exporters
world construction and layout
scripted event creation
physically-based audio/animation arrangement
Figure 3. Single-player 3D game from 2004 [4]