An Architecture For Game State Management Based On State Hierarchies

Luis Valente Aura Conci Bruno Feijó*

Universidade Federal Fluminense, Instituto de Computação, Niterói, Brazil
VLab/IGames, Dept. of Informatics, PUC-Rio, Rio de Janeiro, Brazil

Abstract

Computer games can be regarded as state machines as far as their stages are concerned. The traditional design for this state machine is to assign *ids* to the states and to use conditional constructs to direct the game execution to the proper state. However, this approach is not robust enough and can quickly get out of control. This work proposes an object-oriented model to state specification and management, which features state hierarchies.

Keywords: game state, design patterns, frameworks

Authors' contact:
{lvalente,aconci}@ic.uff.br
*bruno@inf.puc-rio.br

1. Introduction

Computer games usually present many different stages. For example, let a hypothetical game be composed of the following stages: an introduction (first state), the main menu, the main game, and a special menu that is accessed in-game. Figure 1 illustrates this situation.

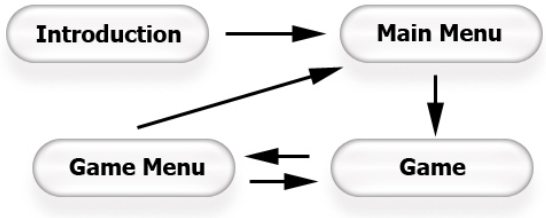

Figure 1: Hypothetical game state machine

The nodes in Figure 1 can be thought as elements of a state machine. This game state machine reflects the current stage that a game is.

A common implementation for game state management, specially with non-object-oriented languages, is to assign identification numbers *(ids)* to the game states and to provide a main function handler to redirect the execution flow to the corresponding game state handler, using conditional constructs such as `switch` and `if/else` [LaMothe 2003]. For example, consider the following pseudo-code:

```
void GameHandler ()
{
 switch (GetCurrentState () )
 {
   case INTRO:  Intro (); break;
   case MAIN_MENU: MainMenu (); break;
   case GAME_MENU: GameMenu (); break;
   case GAME_MAIN: GameMain (); break;
 }
}
```

This approach is not robust enough and some of its problems are:

- The main function handler can become very long and difficult to maintain as the number of states increases;
- The game state handlers potentially replicate the game loop implementation, making it difficult to change it later;
- The implementation details of the game states can become scattered throughout their handlers, increasing their complexity and maintenance cost.

This work presents an object-oriented approach to game state management, that applies design patterns in order to have states as well defined entities. Also, this solution features state groups which makes it possible the use of state hierarchies. The proposed architecture is designed considering single-player games.

This proposed solution is implemented as part of the Guff application layer [Valente 2005], a game development tool that serves as the context for this work.

2. Related Work

Other authors such as Lewis [2003] and Larocque [2001] propose object-oriented solutions similar to the one delineated in this article. They apply the state pattern to model the game states, although Lewis [2003] does not explicitly state it.

The solution proposed in [Larocque 2001] uses a game manager class to keep track of the current game state class, as the solution in this work does. The state classes derive from a common abstract class, and are designed as singletons. The state transitions are designed in a slightly different way than the method proposed in this paper. Their game manager interface presents a method to change the current state, which accepts object instances.

The solution by Lewis [2003] features an internal state stack that is similar to mechanism proposed by the present authors. That solution implements temporary and definitive state transitions.

However, the solutions proposed by Lewis [2003] and Larocque [2001] do not take into account state hierarchies.

A slightly different approach is found in [McShaffry 2003]. There, the author presents a simple cooperative multi-task system design around a process concept. A process is described as an execution unit, and is designed as a pure virtual class with a single method named `Update`(). This design does not pose any particular architecture to the game. Inside the main game loop, all processes get a chance to run their `Update()` method. However, this concept is excessively general and is at a lower-level than game states.

3. The Guff Framework

The Guff framework provides a reusable architecture for game development, being a pioneer work at the Universidade Federal Fluminense graduate program. This tool is divided into two main modules: the application layer and the toolkit. The toolkit comprises sets of ancillary classes which handle visualization, automatic management of third party libraries, application configuration, input devices, Math, audio, and utilities. The application layer is the main subject of this work. Currently, this tool is an open source project [Guff 2006].

3.1. The Application Layer

The Guff application layer is modeled as a state machine. The motivation is to have a state corresponding to each game stage (or level), in order to ease their development and management. Figure 2 illustrates the Guff application layer UML diagram. This model presents simple states (the `State` class) and state groups (the `StateGroup` class). The state groups are an alternative to gather together states that possess some kind of relationship.

In order to ease the usability of this system, the framework automatically binds parent and children states. As an example, the following code snippet applies:

```
class State1 : public State
{
public:
 State1 ()
 : State ("state 1") {}

 ...
};

int main ()
{
 State1 s1;
}
```

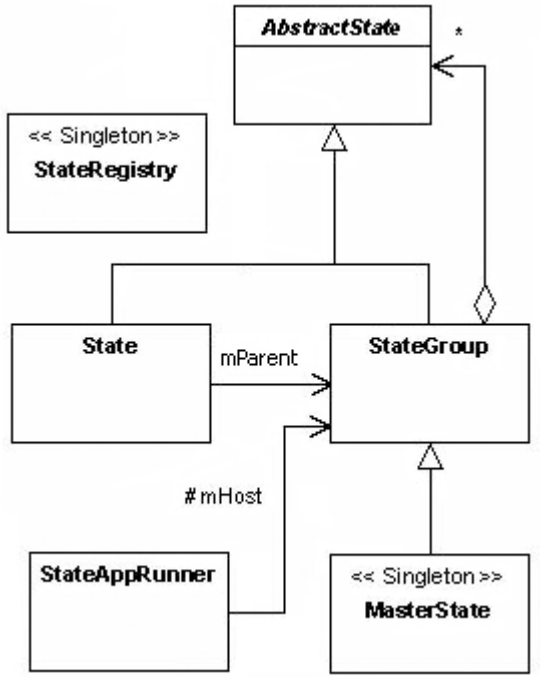

Figure 2: The Guff application layer class diagram

In that example, the code creates a concrete state named "state 1" and registers it with the framework. All states in the framework must have a parent. The framework has a default state represented by the `MasterState` class, which is the root of the game state hierarchy. Whenever the developer does not specify a parent state (like in the example), the framework assigns the master state as its parent. Figure 3 displays the resulting hierarchy.

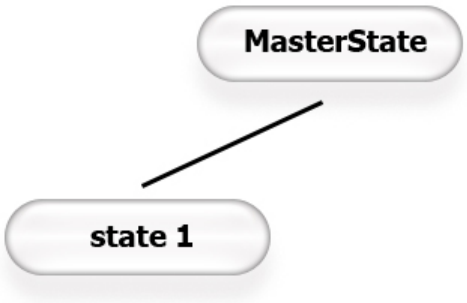

Figure 3: Hierarchy for state 1

If it is necessary to specify another parent, it suffices to inform its name to the framework as in the following example:

```
class State2 : public State
{
public:
 State2 ()
 : State ("state 2", "parent") {}

 ...
};

int main ()
```

```
{
 ...
 State2 s2;
}
```

However, the framework requires that any referenced state must have been created previously in order to be used. A possible approach to solve this situation is to use object composition:

```
class Parent : public StateGroup
{
public:
 Parent ()
 : StateGroup ("parent") {}
private:


 State2 s2;
 ...
};

int main ()
{
 ...
 Parent p;
 ...
}
```

Figure 4 illustrates the resulting hierarchy. State groups have the master state as their parent by default, just as simple states. In a similar way, if it is desired to specify another state as parent, it will be necessary to inform its name in the child state's constructor.

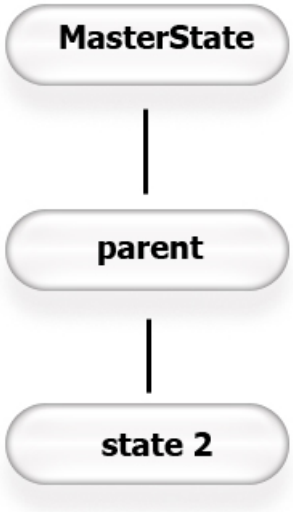

Figure 4: Resulting hierarchy

3.2. Application Layer Details

The application layer UML diagram is conceived as a combination of design patterns. Design patterns describe common solutions to problems that arise in many systems. The solutions are described in a implementation independent manner so they can be applied in different contexts [Gamma et al. 1995].

The design patterns applied in this solution are the State and Composite patterns. The State pattern makes it possible for an object to change its behavior whenever its internal state changes. The object then seems to have changed its class [Gamma et al. 1995]. The Composite pattern helps to build tree hierarchies, so their clients are able to treat simple and composite objects in the same manner [Gamma et al. 1995]. The Guff application layer embodies the State pattern to represent a game state as an specific object. The Composite pattern is applied to have the state machine handle simple and composite states in a uniform manner.

The `AbstractState` class is the common interface for all states in the application layer. This class defines operations categorized into three groups: system events, state events, and events related to the game loop execution model. Derived classes implement those events in order to customize their behavior.

The system events are defined as follows:

- `OnWindowResize`: Dispatched whenever the main application window's size changes;
- `OnMouseDown`: Dispatched whenever the user presses a mouse button;
- `OnMouseUp`: Dispatched whenever the user releases a mouse button;
- `OnMouseMove`: Dispatched whenever the user moves the mouse;
- `OnKeyDown`: Dispatched whenever the user presses a key on the keyboard;
- `OnKeyUp`: Dispatched whenever the user releases some key on the keyboard;
- `OnActivate`: Dispatched whenever the application becomes active (gains focus);
- `OnDeactivate`: Dispatched whenever the application becomes inactive.

The state events are categorized as follows:

- `OnInit`: Dispatched at application startup, on behalf of all states. This is an opportunity for all states to perform whatever initialization is required for their operation;
- `OnShutDown`: Dispatched at application shutdown, so the states are able to release their resources back to the system.
- `OnEnter`: Dispatched whenever the application enters the state;

- `OnExit`: Dispatched whenever the application leaves the state.

In order to describe the events related to the game loop execution model, it is necessary to define what an execution model is. A real time game loop execution model is an approach to manage the execution of tasks present in a game [Valente 2005]. The way the game loop is implemented determines how the game runs in different machines. The Guff state machine applies a Fixed-frequency Uncoupled Model [Valente et al. 2005], so it dispatches events related to the stages of that model. The `StateAppRunner` class is the one responsible for running the game loop. Figure 5 illustrates the Fixed-frequency Uncoupled Model.

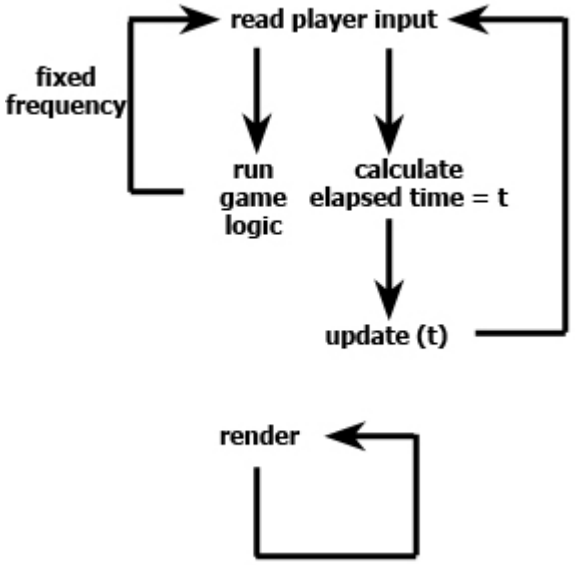

Figure 5: The Guff game loop execution model

The events related to the game loop are:

- `OnFixedFrequencyUpdate`: Represents the fixed frequency update stage;
- `OnUpdate`: Represents the update stage that does not have timing restrictions. It is dispatched as fast as possible and the state receives the time elapsed since the last time this stage was executed;
- `OnRender`: Dispatched whenever it is necessary to render an image. This stage runs as fast as possible, and automatically performs double buffering on behalf of the application.

3.3. State Registering

The `StateRegistry` class serves as a global state repository, which the framework queries in order to automatically bind parent and children states. Its main responsibility is to map names to state instances. The framework requires that the states have unique names for this reason. This class applies the singleton pattern [Gamma et al. 1995] to ensure that only one instance of the `StateRegistry` class exists in the application.

The parent to child binding is performed like in this example:

```
State::State (const string & id, const
string & parentId)
 : AbstractState (id),
   mParent (StateRegistry::Instance()-
>getGroup (parentId))

{
 ...
 mParent->add (this);
}

StateGroup::StateGroup (const string &
id, const string & parentId)
 : AbstractState (id),
   mParent (StateRegistry::Instance ()-
>getGroup (parentId)),
...

{
 ...
 StateRegistry::Instance ()->addGroup
(this);

 if (mParent != 0)
  mParent->add (this);
}
```

The `mParent` object represents a reference to the state's parent. The states may use their parents to request services such as state changes. An object representing a state group is automatically registered with the `StateRegistry`. The only state that does not have a parent is the `MasterState` singleton.

3.4. State Transitions

The states themselves request state transition operations to their parents (which are state groups). The state groups offer two ways of state changes: definitive transitions and temporary transitions. This work defines definitive state transitions as operations that can not remember the state that was current before the operation.

The class interface reflecting those operations is defined as follows:

```
class StateGroup
{
 ...
public:
 // definitive changes
 void changeState (const string &
stateId);

 // temporary changes
 void pushState (const string & stateId);
 void popState ();
...

};
```

Those operations accept the name of the state to which it is necessary to change.

The state groups organize their children states internally with a stack, making it possible to implement temporary state changes. The group's current state is the one on top of its stack.

When a temporary state transition is requested, the state group pushes the current state into the stack and performs the operation. Next, when the state finishes its purpose, the group is able to restore the former state by popping it from the stack. Definitive state transitions do not save state information into the stack.

3.5. State Transition Implementation Details

Figure 6 illustrates a state hierarchy that the remainder of this section will use to describe state transition operations.

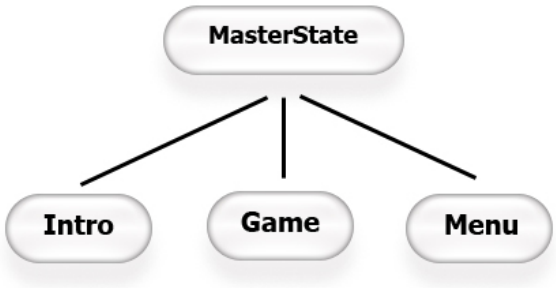

Figure 6: Example hierarchy

In this example hierarchy (Figure 6), the “Menu” state represents a state that may be invoked at any time. The “Game” state represents the main game and the “Intro” state represents an introductory animation that is played at game startup. The “MasterState” is the framework master state. The following code snippet demonstrates some methods from those classes:

```
void Intro::OnUpdate (float time)
{
 totalTime += time;

 if (totalTime > 10)
 {mParent->changeState (“Game”); return;}

}

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

 if (command == MENU)
 { mParent->pushState (“Menu”); return; }
 ...

}

void Menu::OnUpdate (float time)
{
 command = parseInputData ();

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

}
```

The `changeState()` method is implemented like this:

```
void StateGroup::changeState (const
string & stateId)
{
 if (theStateIsMine (stateId) )
  doChangeState (stateId);
 else
 {
   if (mParent != 0)
    mParent->changeState (stateId);
   else
   // invalid transition, an error is
   // reported
 }

}
```

The state group first verifies if the requested state is its child. If this is true, it performs the state transition through the `doChangeState()` method. If the state is not its child then the request is forwarded recursively to its parent. If the request reaches the top of the hierarchy and remains unfulfilled, the system reports an error.

It is important to notice that this design permits state transitions only to sibling and upward states. Figure 7 depicts an example of this situation. In that hierarchy, it is not possible to change from the “e” state to the “h” state. However, it is possible to change from the “e” to to the “c” state.

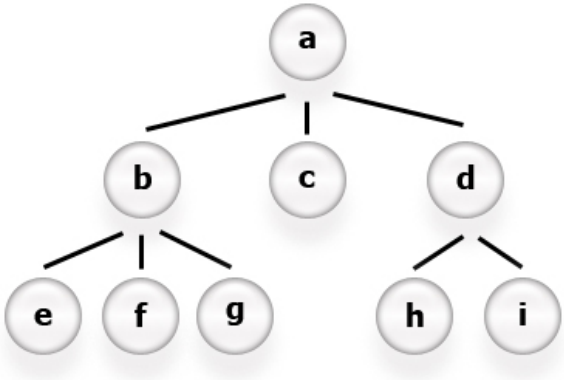

Figure 7: Another example hierarchy

This restriction is due to the fact that the framework considers level one states (immediate master state children) as the application main states, and the remaining ones as their refinements. This assumption simplifies the implementation of state transition operations. The following code snippet demonstrates the implementation of the `doChangeState()` method:

```
void StateGroup::doChangeState (const
string & stateId)
{
 if (stack is not empty)
```

```
  (top of the stack)->OnExit ();

 emptyStack();
 put stateId on top of the stack;
 (top of the stack)->OnEnter ();

}
```

The temporary state transition implementation is similar to the definitive state transition. This operation is defined as `pushState()`:

```
void StateGroup::pushState (const string
& stateId)
{
 if (theStateIsMine (stateId) )
  doPushState (stateId);
 else
 {
   if (mParent != 0)
    mParent->pushState (stateId);
   else
    // invalid transition, an error is
    // reported
 }

}

void StateGroup::doPushState (const
string & stateId)
{
  put stateId on top of the stack;
  (top of the stack)->OnEnter ();
  ...
}
```

There are two differences between these two kinds of operations. Firstly, temporary state transitions do not invoke the `OnExit` event of the current state before storing the new state in the stack. Secondly, temporary state transitions do not flush the state stack.

Last but not least, the state pop operation is defined like this:

```
void StateGroup::popState ()
{
  (top of the stack)->OnExit ();
  pop the current state from the stack;
  ...
}
```

This operation does not invoke the `OnEnter` event on behalf of the former state, which is the opposite behavior of the definitive state transition operation.

4. Example

This section presents an application developed during the Animation and 3D Games course at PUC-Rio that applies the architecture proposed in this paper. The application was inspired by the classic Arkanoid game. Figure 8 depicts a screen capture from the demo.

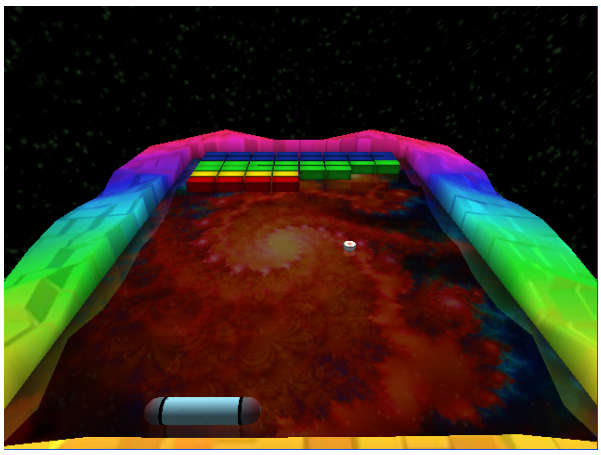

Figure 8: Demo screen capture

The application consists of a state machine that contains three states: introduction, wait for play, and playing. Figure 9 illustrates the state hierarchy.

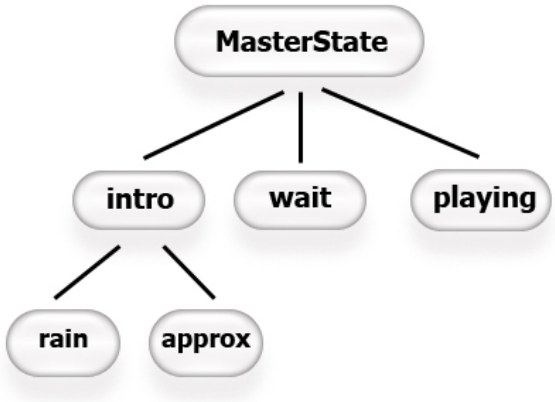

Figure 9: Demo state hierarchy

The demo introduction is a procedural animation inspired by the popular movie “The Matrix”, and is divided into two parts: raining and approximation. The first substate consists of a raining animation just like the movie. The second substate animates the author's names towards the viewer. Figures 10 and 11 illustrates these substates, respectively.

Figure 10: Rain substate

Proceeding from the introduction, the game enters the waiting state, where it waits for the player

command to start the game session. While in this state, the game rotates a camera around the game table, indefinitely.

Figure 11: Approximation substate

When the player signals to start the game, the state machine enters the playing state. The game returns to the waiting state when the playing game level is over. Figure 12 displays the playing state where the player is using the ball camera.

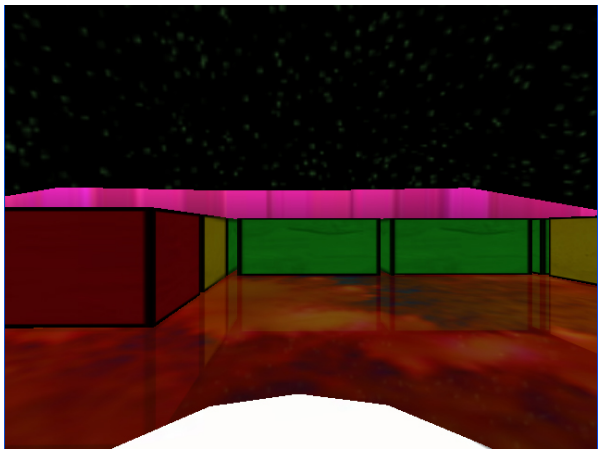

Figure 12: "Ball" camera

5. Conclusion

A computer game can be divided naturally into a set of states. However, there are not many academic works that comprehensively regard this problem.

The traditional solution to this problem is to weakly qualify the game states as identification numbers *(ids)* and to provide a global handler that is responsible for guiding the game execution flow to the proper game state. This solution is not robust enough and does not scale very well.

This paper presents an object-oriented architecture to model and implement game states with a well-defined scope. The proposed architecture builds upon other works by featuring simple states and composite states, thus presenting a state hierarchy.

This architecture considers children states as refinements of main game states. Hence, the composite states may use other states to group related functionality. The composite states present operations to change, push, and pop states.

The implementation details of this architecture are part of the Guff application layer, which is now an open-source game development tool.

Acknowledgments

The first author would like to thank Bruno P. Evangelista and Daniel Monteiro for reviewing this work, and the financial support of CAPES. The second and third authors thank CNPq, and the third author also thanks FINEP for the financial support to this research work.

References

LaMothe, A., 2003. Tricks of the 3D Game Programming Gurus. Indianapolis: Sams Publishing.

Valente, L., 2005. *Guff: Um sistema para desenvolvimento de jogos*. Master's thesis, Universidade Federal Fluminense [in Portuguese].

Guff Project, 2006. *Guff project home.* Available from: http://guff.tigris.org [Accessed 20 August 2006].

Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1995. Elements of Reusable Object Oriented Software. Reading: Addison-Wesley.

Valente, L., Conci, A. and Feijó, B., 2005. Real-time game loop models for single-player computer games. *In: Proceedings of the IV Workshop Brasileiro de Jogos e Entretenimento Digital, 23-25 November 2005 São Paulo. Porto Alegre: SBC, 89-99.*

Lewis, T., 2003. *Managing game states in C++*. Available from: http://gamedevgeek.com/tutorials/managing-game-states-in-c/ [Accessed 20 August 2006].

Larocque, D., 2001. *State pattern in C++ Applications.* Available from: http://www.codeproject.com/cpp/statepattern3.asp [Accessed 20 August 2006].

McShaffry, M., 2003. Game Coding Complete. Scottsdale: Paraglyph Press.