Pigeoncide project

Analysis of results

Juan Pedro Bolívar Puente, Alberto Villegas Erce

2 de febrero de 2010

Índice

1. Introduction

This document explains the arquitecture and features of the project for the Introduction to Game Development course which took place in the first semester in the Turun Yliopisto.

The game description was given in the proposal that we sent to the evaluators in the firs week of the course, so we will move directly to describing the given solution.

2. How to read the source code

The game source code is in the `src` folder of the distribution package. Only a one-liner main file is included in that folder, the rest of the code is included in subdirectories. To ease evaluating the source code –and advancing an overview of the architecture– we describe what is in each of these folders, which in a way are layers of the architecture. Trying to follow the layers from bottom up:

base This module includes basic facilities that are common to any possible software application, such as generic implementations of design patterns, command line argument parsing, configuration storage and loading, etc...

core This module includes core facilities that are common to any possible game application. This includes the process managers, state managers, basic Panda3D management, timers, etc.

phys A thin and incomplete wrapper on top of ODE.

ent This include the entity system, providing a bunch of game independent entities. We discuss entities further later.

game The Pigeoncide specific game implementation.

menu The Pigeoncide menu system.

app The Pigeoncide concrete app facilities.

test Unit tests for many of the modules.

3. Game architecture

The whole project was writen in Python ($2{,}2 < version < 3$) using the Panda3D[**?**] game framework. Panda3D is a very featured system, including a whole bunch of interesting features such as a physics engine –actually two of them, a simple one and an ODE[**?**] wrapper– to a scene graph 3D engine and sound systems.

However, we soon found that its design didn't really fit hour expectatives, at least when it comes to architectural aspects. They can not be blamed for that

though, as their objective is not to provide a well engineered architecture, but a very featured scripting facility that resembles more a *domain specific language* built on top of Python than just a game development framework[1]. Their purpose is to have small game scripts coded fast, leaving features such as scalability and manageability of the code apart.

3.1. Resource management and hierarchical game logic

One of the biggest problems that we saw in the Panda3D system is the insistence on using global state. Even worse than that, the global state was used most of the time implicitly, by using either the global variables installed as Python built-ins, or using many of the facilities of `DirectObject`, that installed event listeners and other entities into global systems via calls to apparently local methods.

Resource management has been and is still one of the biggest problems in computer programming and that kind of practices does not help at all. Sadly, the belief that *garbage collection* has vanished all resource management problems has not done anything else than making the problem worse, as leaking memory is not the worst problem of resource management, and it is easy to see badly programmed Java applications leaving a system blocked because it was leaking SQL connections[2].

Maybe biased by our deep experience with languages supporting the *Resource Adquisition Is Initialization* idiom[3] is that hierarchy is a good thing. This hierarchy, can be made explicit by the means of *lexical scope* –such as in the RAII case– or class design such as the *composite*[**?**] pattern. Panda3D properly implements this in the scene graph.

We take this one step further and make other basic game engine sub-systems hierarchical, providing easier means to control resources, these are, the `task` system, the `event` system, and our `state` system. Let's discuss them further now.

3.2. Abstracting the game loop

Most computer games share a common basic structure: a game loop that iterates gathering the use input, updating the state in response to the input and the previous state, and updating the output to reflect this changes. This game loop iterates around many times per second –usually 60 or so.

For this purpose we implemented a `task` module that abstracts the game loop. This is a well know game pattern and described in books such as [**?**], so we will not step deeply in its description. The important question here is *why*

[1] This explains why they *abuse* Python features and insist on injecting their global variables –which should be punished anyway– into the `__builtins__` module.

[2] No offence, but Java seems to be a honey pot for resource leaks, probably because of its attraction to frustrated C/C++ programmers looking for the relief of garbage collection.

[3] We will not say names, because the lecturers made their language preferences clear during the course ;)

did we re-implement a feature that was already in Panda3d?. We answer the question here:

1. Because of resource management. Many entities in the game may generate many tasks that have to be executed. However, the task system provided with Panda3D is flat, and therefore one could not do such a simple thing as "pause all the tasks belonging to this object". The solution for us was to have a hierarchical task manager, where one object could hold its own local task manager. This way, for example, using `pause ()` on the local task manager will pause all its child tasks.

   We had an interesting discussion about this in the Panda3D with one of its main developers[**?**]. There, when asked about the convenience of re-implementing the task manager the Panda3D developer argued that we would loose nice features that they implement such as the graphical task browser that he found very useful in the debugging process. We finally decided that a hierarchical task manager would not leak tasks and therefore would not need such a tool, and this has proven to be true. When asked about how to group related tasks together, he suggested giving them a name such as `''parent-taskname''` and removing them by using blobs like `''parent-*''`. The fact that Panda3D implements such things as finding tasks using blobs clearly reflects a feature overkill that could be made not needed by using a proper design[4].

2. Many features of their task system seemed to heavy for us. Every task is identified and managed through a string. Also, they where ordered using priority queues, what makes adding and deleting tasks logarithmic. We wanted to be able to abuse the task system adding and deleting them all the time –preferably in constant time– and we did not find the need for string identification for them, etc.

So we came up with a nice and small task system that is the basis for all the game, and can be further studied in the `core.task` module in the source package of our game. In our design, a `TaskGroup` is the task manager, but it is also a `Task`, so they can be hierarchically composed easily. Like in Panda3D, simple functions can be used as tasks too. We also provide a bunch of utility tasks in that module that, combined with lambda's and such make it really easy to write complex animations and interactions in a one-liner –such as `task.sequence`, `task.parallel`, `task.wait`, `task.fade`, ... These utility tasks cover most of what Panda3D call *intervals* and are very easily implemented thanks to our design. Once again, one could argue that Panda3D intervals are more featured, but I argue that most of those features are not needed ;)

Still, we believe that our system still have space for improvements. The main interesting features that we can think of are the support of multi-threading, and optional support for explicit task ordering as done by Panda3D by assigning

[4]Re-reading this, it can seem a bit harsh... I am not arguing that Panda3D developers are unable to come up with proper design, and I really believe that they have done a great job.. Once again, I feel that those kind of features are very useful for unexperienced developers willing to make quick-and-dirty games, which is the main target audience of Panda3D.

priorities. Also, it would be nice for some kind of turn-based games that run on battery based devices to have non-busy waits.

3.3. The Hollywood principle: signals and events

The so-called *Hollywood principle* states that "you should not call us, leave your script and we will call you." That is the essence of modern object oriented programming and the Model-View-Controller architectural pattern that this game implements. In this, the view leave a *reference* in the model, so they get notified when the controller alters it. To achieve this, most of the time the *observer*[**?**] and *multicast*[**?**] design patterns come handy.

Slot-signal libraries usually provide a generic implementation of the pattern[5].

3.3.1. The `base.signal` module

Because we knew before hand that we where going to use *observers* all along the project, the first thing that we started to write just after the course began was the slot-signal mechanism –along with the rest of the `base` module.

A *signal* is an object that represents an event that can ocurr. This is opposed to the event mechanism proposed in the course and used by Panda3D, where events are string identifiers and all go through the event manager. A *slot* is an object that represents a listening end on the signal. When the signal is invoked using either the `notify ()` method or the convenient overloaded `__call__` operator, the signal calls all the listening slots, passing additional parameters if required. Slots are usually arbitrary function objects.

The main problem with the signal mechanism is that they add complexity for the resource management issue that we have discussed all along the way, because it favours setting up complex object meshes that can be hard to track. For this reason, we have implemented a bunch of different facilities that ease this task, such as the `weak_slot` and `slot` decorators, and the `Tracker` and `Trackable` classes implemented in `base.connection`. Also a `base.observer` modules provides easy means to generate whole signal based interfaces for emitters and listeners, which helps in the task of wiring signal connections where we are interested in a whole interface. The implementation was very educative as we used it heavily uses metaprogramming and advanced python features such as data descriptors, decorators, and MRO based collaborative methods.

[5]Pattern orthodoxes find this statement a bit awkward, as patterns are not reusable pieces of code but reusable pieces of design. However, it is a fact that modern computer languages can implement patterns with reusable code; a illuminating masterpiece in this topic is Alexandrescu's [**?**]. Also, some design patterns are often implemented or superseded by language features, specially in dynamic languages such as Python, but also we can see this tendency in C# delegates –observer pattern– or Scala object-class –singleton pattern.

3.3.2. Back to the Panda3D event system

As we said, Panda3D uses a different approach to this, with an event manager that dispatches all the signals, that are identified by string. One can listen on a concrete signal telling it to the event manager. All the basic input handling is done in this way by Panda3D.

The main problem, once again, is that this event manager, called a `messenger` in Panda3D, is global. We wanted, once a gain, some form of scoping and hierarchy. We wrote a similar facility to Panda's messenger, the `EventManager` in `base.event`, that, while missing once again some unneeded features, was able to forward all messages to other event managers. This way we got the needed hierarchy: one can have its own event manager in his "local" that still can be used to receive all its parent signals, while still being able to, for example, mute all its local subscribers. Also, this system mapped the event system to the signal mechanism, providing further convenience.

The only problem here was that Panda3D does not have the possibility to have a "catch-all" method for its events. This was surprising because that is one of the few advantages to have a event manager instead of independent signal objects. To be able to forward Panda3D's events to our event hierarchical system we only could modify the Panda3D implementation. Because we wanted the game to be compatible with third party distributions of the library, this was a no-no. But then Python dynamism comes with an ugly but convenient technique in this paradox: *monkey patching*. This is, modifying a class or object's implementation at runtime. This is implemented in the `core.messenger_patch` module, which should be loaded *before* any other Panda3D module.

This has an important drawback: our game depends on Panda3D's Messenger class implementation, and internal changes on it will break it. It seems to work properly with version 1.6.X and 1.7 thought.

3.4. Wiring everything up

Specially in this MVC context, we end up with a mesh of objects that communicate with each-other, in their role of either model state, view, or controller. Someone else, a *mediator*[**?**], have to wire these things up.

On the other hand, we have been talking all the time that we have carefully implemented hierarchical object structures in order to achieve less coupling and local changes to certain *scope*. The question is, then how to define the scope of a game entity.

For these two purposes the `core.state` module comes into hand. There we have a *StateManager* and *State* class. The state manager provides a stack based state machine implementation. A state can be something like a menu screen, a game, a loading screen, etc. A game itself can be divided into different states, if needed. A state is notified whenever it is no longer the top of the state stack, and when it is again the top of the stack, for example. States, combined with local managers, enable us to implement trivially otherwise not-so-easy features

such pausing the game, having an in-game menu, etc. Every state provides its own task manager and its event manager.

The whole application is driven by states. The basic application framework is abstracted in the `base.app` and refined to work with states in `core.app`. From there onwards, one just register states and moves around them by changing the current state, entering a sub-state, leaving a state...

Also, the states are lightweight enough to be used to implement other state machine based features such as AI. Still, there is a to-do note in the state module talking about refactoring states to have an even lighter version for such purposes in the future...

3.5. Entities, a place for experimentation

What is an "entity"? An entity is an object inside the game world. That is all what we can say about it. But, as for concrete entities, we can say that:

1. They can have a lot of orthogonal features. For example, they can be a physical entity that reacts to its environment. It can be an entity represented visually by a 3D model, or an animated actor, or a 2D texture. Or it might be a physical entity, but that stands all the time, such as the player does. An infinite etcetera follows...
2. When writing a concrete entity, or an orthogonal feature for an entity, we want to concentrate on the logic that drives the entity or the feature that it adds, and not on boilerplate (a) resource management or (b) object wiring.

During the course, the lecturers talked about event-driven architecture, and at some point also about how things can actually get messy –i.e. the example about the interactions between the physics system and the scene graph. Also, while developing the `base` module had been learning a lot about advanced Python features, and some background voice was asking for getting rid of all that event-based thing to try to explore something... else. Then, we had access to the BHive source code, the event based game engine developed by the lecturers of this course, in order to find some inspiration.

The `world.WorldLinkers` module convinced me about trying a different approach. The whole thing seemed too static for a dynamic language, we thought.

The fact is that events, as used by BHive, provide a very nice way to keep features orthogonal, as stated in our first requirement, but fail in the second. What we propose in our experiment is something different: using a language feature, *multiple inheritance*, to combine these orthogonal features of a single entity. MI! That is evil! Well, not that much...

In (good) dynamic languages like Python, there is a nice feature usually referred to as *call-next-method*, after the name of the function that implemented it in the Common Lisp's object system. In Python, this is implemented using the `super` built-in that returns the proxy of an object that calls the next method

of a class. This feature allows to call all the methods in a multiple inheritance environment in a consistent order. This consistent order is a linearization of the class hierarchy that satisfies the following conditions:

1. If a class `Derived` inherits from a class `Base` then `Derived` comes first.
2. If a class `Derived` inherits form both class `BaseA` and `BaseB`, the `BaseA` also comes before `BaseB` –this is, there is left-to-right order in the same level of a hierarchy.
3. Every class appears only once in the linearization.

Methods designed to work in this way –this is, they pass the control to the *super* class, where the *super* is defined in the previous terms[6]– are called collaborative, because they collaborate with *unknown* nodes in the class hierarchy. This feature is only possible in dynamic languages, because the concrete type of *super* inside a given class can change when other classes derive from it[7]. Much further can be said about how to program using this technique, but this is not the purpose of this text.

What is to be said, is that we use this to wire, implicitly, using language features, the connections between these orthogonal facilities of a game entity. When one of this feature depends on some other features, it derives from them. The C3 MRO ensures that these dependencies will be correctly preserved even if the hierarchy gets very complex, so no coupling is produced with other unrelated features.

By using this, we solved problem 2.b. Also, to solve 2.a, the entity system comes handy. The resources required for a given entity are given by its entity manager. Every created entity is registered into the entity manager that it receives in its constructor. Note, that entity managers can also be combined using this mix-ins technique. There is also a `GameState` that specialises `State` adding a entity manager that is set up with entity manager able to hold visual an physical entities at a local scope, something that is achieved, for example, by allocating its own node in the scene graph. The entity manager is usually also in charge of keeping track of all the entities for easy cleanup when we leave the state, and it manage the entity resources; i.e. the `PandaEntityManager` allocates a new node for the entity in the scene graph on its creation, and removes it on its disposal. This can be sometimes done with the collaboration of the entity itself.

Most of game independent entity facilities are developed inside the `ent` submodules, while many specific ones are implemented in the `game` module. We are very happy of how with achieved to minimise boilerplate and maximise power. Still, some things can be improved and the design is a bit experimental. The implementation is plagued of to-do notes which are loud thoughts on how to improve the design or the implementation. Much more can be said about this system, for example, what *decorators* and *delegates* are in this context, and how

[6]The C3 Method Resolution Order. Further literature can be read here [**?**].

[7]Still, we have an idea for some approximation for C++ in mind using heavy template metaprogramming...

they are used to provide "dynamic" entities –i.e. facilities that can be added and taken from an living object. But this text is getting too long, so we leave the code as a testimony of it, and we are open to have discussions about this with the evaluators of the course in person :)

4. Implemented features

We implement most of the features proposed in the course. We describe in the following which of them and how.

4.1. Hierarchical scene graph

We use Panda3D hierarchical scene graph in all our graphical code. All the entities into a game state are child of a sub-node of `render` –this is, only the one node per state is in the `render` root node to avoid its pollution. Each entity has its own node, with the current state node as its parent. Some other sub-nodes are used to tune the relative position of the parts of an entity. We also use further features of the scene graph, such as attaching the node of a weapon to the "hand" joint of the skeleton of the animated boy, and properly using our entity combination system to inject its coordinates to the physics system, etc.

4.2. Physics

We use the ODE wrapper distributed with Panda3D as physics engine. We must say that this has been quite painful, for the following two reasons:

1. ODE is very bad at solving the tunneling effect. This means that when the frame-rate drops, tunneling happens a lot, specially for fast entities. This was specially a problem for the pigeons, but we partially solved the problem by manually sweeping its collision volume. The collision geometry of a pigeon was initially a sphere, but we changed it to be a cylinder that we use as a swept sphere whose length depends on the distance that the entity has moved in this frame.

   Still, the problem sometimes persists for other entities at low frame-rate and fast movement conditions. One solution is to try to keep the simulation running at a different frame-rate than the rendering, but this is less trivial as it might seem...

2. Either ODE or the Panda3D wrapper is plagued of bugs. When doing many apparently inoffensive things, but that ODE dislikes, it either segfaults or crashes the application with just a not-very-informative message. When debugging your application this makes it really a mess, because there is no way to backtrace the segmentation fault or ODE error to the Python code. Sometimes one can get a small backtrace of the C++ of the wrapper executing Python inside `gdb`, but there is no way to know at which point of the Python script the problem occurred.

Sadly, there is no Python wrapper for Bullet physics[**?**] apart from the one embedded in the Blender Game Engine, so we do not have a replacement for ODE in the near future.

4.3. Event-based game logic

We have not been very orthodox in implementing this, as explained in section **??**. However, our alternative architecture is as capable or more than an event-based one. Also, the fact that we did not use to couple together the different *faces* of a world object does not mean that we did not use events.

As we described in 3.3, we have put a lot of effort in implementing the generic low level components that are the basis of a event based architecture. And we use events, all the time. They are used to wire up most of the game logic. An example, when a physical entity collides, it signals a `on_collide` event, that is listened by the *killable* entity that it belongs to, that in turn shows some particle effects and sounds, and then triggers the *on_death* signal, that in turn is listened by a function that finishes the game and enters the game over state for the player entity, or if the entity is a pigeon is listened by a function that increases the counter of dead pigeons and by another function that checks whether there are more pigeons in the level to maybe enter the game win state.

Also, while we did not go very further in this in our previous description, while *collaborative methods* are used to communicate the different aspects of a single entity, events are used to communicate among unrelated entities. Actually, an entity can be *observable* if it dispatches events on its spatial properties updates. In this way, you can make a pigeon follow the boy or follow some piece of food by changing no line of code apart from one that changes which entities events the pigeon listens to. The same happens for the camera, and so on.

We even use events for the configuration system. Whenever a configuration node is changed a signal is dispatched, allowing unrelated listeners to be updated depending on the configuration state –for example, when the music-volume option is changed, the menu can listen to update the position of the scrollbar that changes the music volume, and an object in charge of managing the Panda3D system changes the volume parameter of the audio manager in charge of playing the music.

So, while having an non canonical –but maybe better?– implementation of our events and entities systems, we definitely have a event based architecture for gluing up almost all the application logic.

4.4. MVC game architecture

A consequence of the event based game logic is that we can easily come up with a MVC game architecture. However, when we introduce the *mixin* based entities it might seem non-obvious how our architecture is MVC. More exactly, how can be the views and controllers of an entity decoupled from its state if they are combined via inheritance?

There are three ways in which this is achieved, and the best one depends on the concrete requirements:

1. First, there is the use of *delegates* and *decorators* system. If an entity mixin provides a delegate version of it, its controller can be dynamically changed using decorators. The decorator implementation is shared with the non-decorator version of the entity mixin, the only difference is that the decorator one controls communicates with its dependencies via a delegate –this is made in a transparent way, thanks to Python's magic.

   The decorator can be seen as a controller that adds behaviour to a "static" model entity on runtime. For example, this code could be possible:

   ```
   pigeon_model = mixin (PhysicalEntity) (entities = ...)
   pigeon_controller = mixin (BoidEntityDecorator) (
       entities = ..., delegate = pigeon_model)
   ```

   One can think that this produces code duplication, because we need to have two different versions of the `BoidEntity` and an alternative controller version. However, the implementation can be shared, and the following pattern arises when developing a controller that can be used embedded with the model or as a separate object:

   ```
   class BoidEntityBase (Entity):
      """
       Implementation of the flocking algorithm, asuming that all the
   needed data is embedded into self.
      """
      ...

   class BoidEntity (DynamicPhysicalEntity, BoidEntityBase): pass
   class BoidEntityDecorator (DelegatePhysicalEntity, BoidEntityBase): pass
   ```

   The delegate is in charge of using *properties* and functions to forward all the functionality that a `DynamicPhysicalEntity` would provide through another object. These delegates by now generate a bit of boilerplate because the forwarding is encoded by hand, by we are planning to adapt the functionality of `base.proxy` to generate it automatically using Python's metaprogramming magic :)

2. Note that the previous code works only for wiring controllers with the model, because there is no way by which the decorator would be notified when the delegate changes, and, as such, we cannot connect a view to the model in such a way.

   But, as the previous code shows, one can build the desired class on runtime mixing the needed entity components. For this, we provide the utility function `mixin` that creates a new class from a set of bases. For example, this code could be valid[8]:

   ```
   if game_is_local:
       pigeon = mixin (BoidEntity, DynamicPhysicalEntity, ActorEntity) (...)
   ```

[8] We do not implement networked game, so some of the classes described are ficticious.

```
elif game_is_client:
    pigeon = mixin (ActorEntity, RemotePhysicalEntity) (...)
elif game_is_server:
    pigeon = mixin (BoidEntity, DynamicPhysicalEntity, ServerPhysicalEntity) (...)
pigeon.set_position (100, 100, 100)
```

In that way, the *call-next-method* mechanism would make sure that when a manipulation method is called on this model+view mixin object it propagates correctly through all the hierarchy notifying the views.

3. However, there is still a problem there. In the original MVC, one should be able to connect the model should be able to notify many views. Also, in the previous example one can not change the connections of the model with the views after the object is built.

   For this purpose, we can fall-back to the events mechanism. This example code shows how this can be made, for example, connecting a follower camera to a pigeon.

```
class FollowCameraEntity (Entity, Trackable, SpatialEntityListener):
    ...

boy    = mixin (Boy,    ObservableSpatialEntity) (...)
pigeon = mixin (Pigeon, ObservableSpatialEntity) (...)
camera = FollowCameraEntity (...)
...
pigeon.connect (camera)
...
camera.disconnect_sources () # Method added by Trackable
boy.connect (camera)
...
# If it was not a trackable we have to remember who it was connected to.
boy.disconnect (camera)
```

   As we can see, we provide a nice way to separate algorithms from the needs of the object topology. They should specify their requirements inheriting from interfaces that are implemented in different ways, and then later combined in runtime.

   Note that because abstracting all needed Panda3D features in our system is a lot of work, and because we sometimes took shortcuts in our code for faster development –like skipping separating interface and implementation...– our system might not always be as flexible as previously shown if taken just outside the box. The architecture not only is MVC, but it can be more or less MVC depending on the concrete needs of our system as previously shown, and only polishing a pair of corners in the code remain.

4.5. Artificial Intelligence

The pigeons movement is modelled using Craig Reynold's flocking algorithm [**?**][**?**]. This allows to model the complex behaviour of the flock of birds using autonomous agents with a set of simple rules. We implemented it in a quite generic way –does not depend on any kind of representation, being easy to combine with any using our MVC architecture.

Also, that is not enough to model the complex behaviour of the pigeons. A state machine is used for this, with many states: fly, walk, follow, fear, eat, hit, land, return, attack. Most of the states just changes the parameters of the flocking algorithm, but some of them do few extra things. The fact that they are implemented using the state manager and our generic flocking algorithm makes the code small and simple.

Sadly, our flocking implementation has an $O(n^2)$ time complexity, being the biggest performance bottleneck of the application, and making the application not very responsive when having a high number of pigeons in the scene –more than 30 pigeons drops the framerate to 20-30 FPS in our computer and the tunneling problem of ODE becomes significant. We are planning to improve this in the future by using proper space partitioning, or even moving the core of the flocking algorithm implementation to C or C++ code.

4.6. Lights and shaders

We do not use many lights in the scene –we don not really need them– but still some to have a nicely illuminated scene.

But we use *glow mapping* to get nicely shaded laser fields and highlighted sticks. We would use them in more places if it were not because of the few problems that we had with the graphics –see the later section.

We implemented the glow mapping using a fragment shader pipeline. First the whole scene is rendered to an off-screen buffer using an alternative texture that is combined with the normal one to determine which regions should glow. Then that off-screen buffer is blurred using two shaders, one that blurs the texture vertically and another one that blurs it horizontally. Then, this buffer is merged on top of the normally rendered scene, resulting in a very nice glow effect.

5. Conclusion