

SPACE SPORTS / TRAINING SIMULATION

Nathan J. Britton
Information and Computer Sciences
College of Arts and Sciences
University of Hawai'i at Mānoa
Honolulu, HI 96822

ABSTRACT

Computers have reached the point where advanced physics simulations can be executed in real time on relatively small and cheap machines. With this equipment readily available, we have an opportunity to provide educational and training simulations that can galvanize the next generation of explorers. The goal of this research is to create a "Space Sports" game for K-12 students that allow them to build and experiment with sports games in different extra-terrestrial environments. By playing these games, the students will come to understand concepts like gravity, and their effects, in terms that students can see and experience first-hand.

To accomplish this, a game-creating engine needed to be developed that allows a 3D graphics rendering engine to work with a rigid-body dynamics engine and a networking engine. There are games that use these narrow engines directly, but that limits the flexibility and expandability of the software. A game engine that allows for modular, swappable components for graphics, physics, and other solutions would be much more flexible. In order for this work to expand and be used for actual training simulations in the future, it was decided that a modular game engine would be developed.

COSMOLYMPICS GAME DESIGN

The code-name for the space sports game itself is Cosmolympics. In order for Cosmolympics to work as a game creator, it is imperative to define what exactly a "game" is in a concise, algorithmic fashion. Considering that the primary purpose of Cosmolympics is to illustrate the effects of gravity, the type of game needed to be narrowed to ball-oriented games. That is, the game and point distribution will always be based on the movement of ball-like objects that are kicked, thrown, and otherwise manipulated in the field. As a result, any change in the gravity of the playing field will have dramatic effects on any game created.

From there, definitions were specified. Terms that are defined appear italicized:

Game - A series of *events* executed by a set of *agents* within an *arena* repeated until a time limit or point limit is met or exceeded.

Event - A change in the position of a set of *artifacts* which satisfies the condition of a *goal* based on the *artifact's phase* and results in:

*A change in *phase* of the *artifact* and/or

*An accumulation of points for the corresponding set of *agents*.

Artifact - A mobile object within the *arena* whose position in relation to corresponding *goals* determines the progression of *events*.

Goal - An immobile object or field whose position in relation to *artifacts* determines the

progression of *events*.

Phase - A list belonging to each *artifact* specifying which *goals*' conditions can be met by that *artifact*. An *artifact* may have multiple *phases*, but only one can be active at any given time.


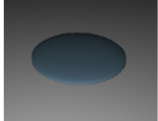
Agent - An autonomous element of the system that effects change on the *artifacts* in the *arena*. May also be an *artifact* itself and effect events based on its location in a *region*.

Arena - The physical boundaries of game-play within which all *agents*, *artifacts*, and *goals* must exist and cannot leave.

Region - A special type of *goal* that has no physical dimension and serves to partition the arena into sub-sections.

These lists of definitions provide a common language with which to address game creation and establish the domain of the game creator. Once they were defined, each of these words could then be made into a "class" in the code wherein the relevant functions and data could be organized. With this, it became possible to imagine specific Cosmolympic games; which allowed for work on the next step - the game creator interface.

It is important in all software that the interface is as intuitive and easy-to-use as possible; considering that the target age group could be as young as elementary school students, it is particularly true in this case. A sand-box-type interface with direct manipulation to move the artifacts in the arena to drive rule creation was determined to be the best choice for ease of use. The player will be able to pick an arena (or optionally design one) and then place goals and artifacts within it, using point-and-click mouse controls. The player would then only have to use the mouse to pick up and toss the artifacts around.

Artifact	Goal
	

-Interaction Type-

Bounce Rest

-Conditions-

Duration of Rest:
 Between s (and s)

Speed of [artifact]:
 Between m/s (and m/s)

Angle of [artifact]:
 Between ° (and °)

Position of other Players
 Region1
 Agent1
 Agent2
 ⋮

Relation to other rules
 Rule 1 with after
 ⋮

-Points-

Points awarded pts
 Team A
 ⋮

Upon a collision between an artifact and a goal, the system will mark the goal and automatically create a condition for it. The system will make this decision based on the type of goal and the type of artifact in question. Figure 1 illustrates the window that will pop up with a confirm/cancel option upon collision. This will give the user some control to correct errors and specify more sophisticated conditions. Figure 1 is an example with a ball artifact and a platform goal.

The interaction type determines which of the conditions below it are available to choose from. If the intent is for the ball to hit the platform and bounce, then the player has the option of specifying a minimum/maximum speed and angle of approach. If the intention of the player is for the artifact to rest on the platform, as in a baseball player standing on a base, then the player could specify an optional duration constraint in the same fashion.

For any goal condition, the position of other players can be set as a pre-condition.

Figure 1: Interface dialog for rule creation.

Similarly, any goal condition, or rule, can be set to relate to any previously created rule. The phases and phase change conditions will be inferred from these rule relations. This is a consideration that allows for sequences of actions that have to be performed in a certain order. For the final action in such a sequence, or for single-actions, points can be awarded.

At this point, it was decided that enough game design specification had been done to warrant implementing an initial prototype.

MODULAR GAME ENGINE

In order to get a prototype of the game running, a set of solutions for 3D graphics rendering, rigid-body physics simulation, and networking were required; in addition, the system would greatly benefit from incorporating a (non-essential) scripting language that interprets code in real-time for quick level-editing. At the time this research was being conducted, there was no all-in-one solution that offered all three of the essential solutions with free access to the source code. There were, however, open source packages that dealt with each of those three respective problems individually: Ogre 3D for graphics rendering, ODE (open dynamics engine) for rigid-body physics simulation and RakNet for multi-player networking. As for the scripting language, at the time of publication, the Python scripting language was the best open-source, interpreted programming language available.

It was therefore determined that a game engine would need to be created from those three component engines with a Python scripting language. The engine that was created is simply a framework that allows multiple pre-existing software to work together in a modular fashion. This is a significant distinction from what is commonly done - that is, creating a game by calling functions from the appropriate module *directly*. When the functions are called directly, it makes the code that is being used very inflexible; when a sub-engine such as Ogre3D has a significant update, or if a new sub-engine with more desirable functionality is released, the game must be re-written to accommodate the changes.

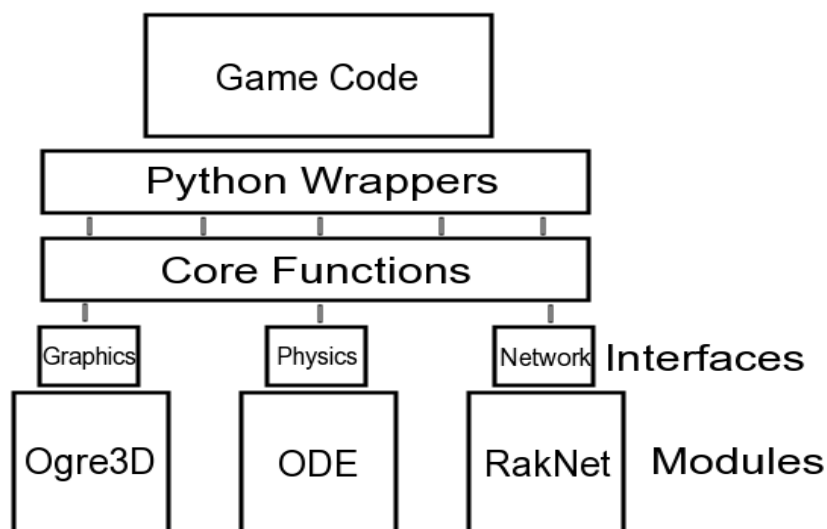


Figure 2: Architecture of a modular game engine.

If the modules are designed to work in a modular fashion, they could be changed and swapped out while the code for the game would not need to change to accommodate it. Such a system would have great potential to reduce re-writing code and allow Cosmolympics to easily expand into larger scale games and training simulations. Object Oriented Programming Languages, like C++, could theoretically achieve this goal through polymorphism. A candidate system was designed to test this hypothesis using the architecture illustrated in figure 2.

The system is separated into five layers. Each layer is only able to access functions from the layer directly below it. The modules are the lowest level of the system, and they are separated from the "core functions" of the engine by a layer of interfaces. The interface layer allows for the creation of these core functions which incorporate, for example, graphics and physics functionality without mixing direct calls to the respective modules.

This is achieved in C++ by what is called an "abstract class". An abstract class is not actually implemented, but still specifies what kind of data and functions it should have. This is essentially pointless, unless there is a class that inherits from it, in which case the child class must implement the prototypes (i.e. abstract functions) of its abstract parent class. Each child class then has the same set of functions with the same names, the same input and the same output. The difference is only in how exactly the functions are actually implemented.

Initially, three abstract classes were created: GraphicsInterface, PhysicsInterface, and NetworkInterface. Then, a specific interface that inherits from the general interface was created for each module. The OgreInterface, therefore, is a (i.e. inherits from the) GraphicsInterface. The result is that a GraphicsInterface object can be instantiated (created) in the core functions layer and the GraphicsInterface functions can be called; but because the actual implementation is handled by the OgreInterface, the core functions can use Ogre without ever touching Ogre directly.

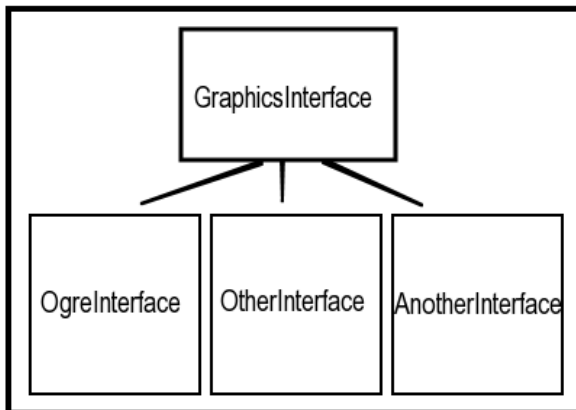


Figure 3: Three child classes inheriting from the abstract GraphicsInterface.

This is a trivial result when there is no other implementation to choose from, which is the situation as of publication of this study. However, when there is another implementation available, as illustrated in figure 3, then it will be possible to test the merits of the modular game engine. Theoretically, if all the interfaces are implemented correctly, it would be possible to change any given engine by only altering a single line of code in the core functions layer.

For each module that is added to the engine, the number of possible configurations would increase dramatically without requiring any work other than implementing an interface. Implementing a child interface is a fairly straightforward process because the abstract interface tightly restricts and directs what needs to be programmed; with some basic familiarity of the module, it can be done quickly. The issue, however, is designing the initial abstract interface, which, contrary to the child interfaces, is not the trivial task that it was initially assumed to be.

Only a very basic GraphicsInterface was specified in the time that it was estimated to take to complete all three interfaces. The decisions regarding function prototypes are very important

and decoupled from specific implementation, making it a time-consuming task. Although considered good practice for programming in general, in this case it is also imperative that each function be accompanied by appropriate documentation to describe exactly what task it needs to perform. Once the `PhysicsInterface` and `NetworkInterface` are complete, a handful of core functions should be all it takes to get an initial prototype of *Cosmolympics* running.

CONCLUSIONS

The game design for *Cosmolympics* is thorough and ready for prototyping. In retrospect, however, for the purpose of creating a simple space sports game, the decision to create a modular game engine was superfluous and distracting. The amount of time it took to get a modular plan and build a fraction of the framework required to preserve modularity delayed implementation of the actual *Cosmolympics* game; which in and of itself does not require modularity. However, the potential merits of the modular game engine are quite promising provided that certain issues can be resolved.

The modular game engine could reduce a great amount of work when upgrading or swapping out engines; this is vital for long-term expandability. However, in order to gain access to this advantage, a significant amount of work needs to be done developing interfaces. The idea is to focus the work that needs to be re-written to a significantly smaller area in the software. A working example needs to demonstrate at what point (# of modules vs. size of game) the benefit exceeds the cost in terms of development time. There also needs to be a working example of a game swapping engines through a change in just one line of code to prove the concept.

Ogre handles 2D user interfaces with the sub-module CEGUI, and input devices (mice, keyboard, joystick) with the sub-module OIS. These need to be separated into fourth and fifth abstract interfaces respectively, otherwise all graphics engines will have to have that capability and it will not be flexible to those ends.

There are a multitude of different types of physics engines. Some, like ODE, deal only with rigid-body dynamics, some deal only with fluid dynamics, etc. Since each physics engine must be capable of providing similar output, the only comprehensible option would be to break the `Physics` interface up into appropriate categories, but until there are more options to choose from, that is going to be a difficult issue to resolve.

ACKNOWLEDGMENTS

I would like to thank Dr. Binsted for pointing me in the right direction and reminding me to tackle design and specification *before* implementation. I would also like to thank the Hawaii Space Grant Consortium for helping to fund my senior year of undergraduate study through the Space Grant Fellowship. I am grateful to Ed Scott and Marcia Rei Sistosio in particular for their support of my attending the NASA MMO workshop to discuss the potential for *Cosmolympics* in an educational MMO game.

This is the beginning of my most ambitious project and the foundation of my long term professional goals in the space industry. I credit the support of Dr. Binsted and the Hawaii Space Grant Consortium for my acceptance to the International Space University in Strasbourg, France, where I plan to continue research in training simulation technology.

Special thanks also to Torus Knot Software, the Ogre3D community, Russel Smith (developer of the Open Dynamics Engine), Jenkins Software (the developers of RakNet) and the Python Software Foundation for providing the brick and mortar.