Creating an Abstract Physics Layer for Simspark

Studienarbeit im Studiengang Informatik

vorgelegt von

Andreas Held 206110227

Betreuer: Dipl.-Inf. Björn Pelzer, Institut für Informatik, Fachbereich 4

Erstgutachter: Prof. Dr. Ulrich Furbach, Institut für Informatik, Fachbereich 4

Koblenz, im November 2010

Abstract

This paper documents the development of an abstract physics layer (APL) for Simspark. After short introductions to physics engines and Simspark, reasons why an APL was developed are explained. The biggest part of this paper describes the new design and why certain design choices were made based on requirements that arose during developement. It concludes by explaining how the new design was eventually implemented and what future possibilities the new design holds.

Introduction

Simspark is a generic physical multiagent simulator. One of its flaws was its dependency on one specific physics engine, the Open Dynamics Engine (ODE). Relying on a single and arguably outdated physics engine held back Simspark's potential compared to what it could do with support for more and better physics engines. However, switching ODE with another physics engine would only postpone the problem, allowing it to resurface if another, even better physics engine is released. Furthermore, relying on a single physics engine hampers Simspark's flexibility. Thus, development of an abstract physics layer (APL) was started. The relevant library within Simspark underwent a redesign, which was eventually implemented.

The APL serves as a mediator between Simspark and the actual implementation of the physics simulation. Simspark does not know which physics engine is used by the implementation. As a result, it is possible to change the implementation without having to change anything within Simspark. The implementation was designed as a plugin. Once more implementations using different physics engines are available, using a different physics engine will be as simple as using a different plugin.

This paper starts out with basic introductions to both physics engines and Simspark. After that, it discusses Simspark's original design, how the relevant library was changed and why these changes were made. The last part covers a selection of problems that occured during development, as well as their solutions, in greater detail and explains the benefits of the new design.

Contents

1 An Introduction to Physics Engines

1.1 The Major Physics Engines

Physics engines are available as both OpenSource and commercial tools. According to a survey by the Game Developer's Magazine, **Bullet**[6] and the **Open Dynamics engine (ODE)**[5] are two of the most frequently used open-source physics engines[6]. However, ODE reached its zenith several years ago. After that point, support for ODE receded and today, development for ODE has nearly come to a halt: The most recent version as of writing this, ODE 0.11.1, was released in October 2009[5]. After ODE lost momentum, Erwin Coumans started Bullet to offer an alternative open source physics engine. Since ODE was also open source, Coumans used its source code as a guideline and as a result, Bullet is strikingly similar to ODE in both structure and function. The most important difference, however, is that Bullet is a lot more efficient than ODE, and the name Bullet was chosen to reflect that: Several users have reported that Bullet does, in fact, perform a lot better than ODE in certain situations, e.g. when there is a large stack of boxes[6]. Support for Bullet is still very strong and new versions are released in regular intervals. Because of that, Bullet also offers a lot more functionality than ODE. Bullet's downside, however, is that it is hardly documented. Because of this issue many users actually refer to ODE's documentation to learn things about Bullet.

Commercial physics engines include **PhysX** by Nvidia[1] and **Havok**[2]. Their main advantage is the availability of development tools that allow users to create simulations as easily as possible. Despite being available for free, ODE and especially Bullet can compare against their competitors in terms of correctness and efficiency. To cut costs, some developers choose Bullet over commercial physics engines. Bullet has been used in some well- known projects, including the video game Trials HD[6] and the movie 2012[6].

1.2 The Hello World example in ODE

The most basic example for all physics engines is a sphere that is located in the air at the beginning of the simulation and, affected by gravity, falls down to the floor. It is a great way to explain the basic concepts behind a physics engine. Because ODE is the main focus of this paper, this section will explain how to implement this brief scenario in ODE.

[1] http://www.nvidia.de/object/nvidia_physx_de.html
[2] http://www.havok.com/

First of all, we must create a `world`. A world is simply a virtual universe that the simulation will take place in. A world has some basic parameters, e.g. the direction and strength of gravity. In this example, we want gravity to run along the Z axis (i.e. the Z axis points up) and be as strong as it is on earth, so we set 9.81 as the parameter. To create a world, ODE offers the method dWorldCreate that returns the world's ID as a parameter. A second method, dWorldSetGravity, takes the world ID as a parameter and also the strength of gravity along each axis.

Next, we need the floor, which is easy to take for granted. The most basic floor is simply a flat surface that is perpendicular to the gravity vector. Unsurprisingly, the function to create a Plane is called dCreatePlane. The first parameter is a dSpaceID; however, we will not create a space in this example, so we will pass zero to indicate that the plane will not be part of any space. We also need to pass four parameters a, b, c and d to dCreatePlane. These are the parameters of the plane equation $a*x + b*y + c*z = d$. The floor's plane equation would be $z = 0$, so we set $c = 1$ and $a = b = d = 0$.

Last but not least, we must create the sphere itself. We use dCreateSphere. Again, we pass zero as the spaceID. The second parameter is the radius of our sphere. A shape like the one defined with dCreateSphere is called a **geom** and is used almost exclusively for collision detection and visualization.

Now, we have a spherical shape with a radius of 1, but it is not part of our simulation yet. We also need to create a **body** with dBodyCreate. dBodyCreate requires the ID of the world that the body should exist in. Naturally, we pass the ID of the world that we created in the beginning to this function. Next, we set the body's position with dBodySetPosition by providing the x, y and z coordinates. The sphere is supposed to be 50 meters above the ground in the beginning, so we set $x = y = 0$ and $z = 50$. Finally, we must tie the body and the spherical shape together with dGeomSetBody. After that, the two of them form an entity that constitutes an object in our simulation. The geom defines the shape and size of that object; the body defines the location, rotation, velocity and mass of an object. Together, the body and geom model a sphere as one would perceive it in the real world. Note that we do not need to pay attention to the mass of the object in this simulation because all objects, regardless of their weight, fall at the same speed. In C, the source code for setting up this brief example would look like this:

```
dWorldID world = dWorldCreate();
dWorldSetGravity(world, 0, 0, -9,81);
dGeomID floor = dCreatePlane(0, 0, 0, 1, 0);
dGeomID sphereGeom = dCreateSphere(0, 1);
dBodyID sphereBody = dBodyCreate(world);
dBodySetPosition (sphereBody, 0, 0, 50);
dGeomSetBody (sphereGeom, sphereBody);
```

All that's left is actually running the simulation. There are several ways to do this. We must tell ODE to step the simulation by providing a time interval in seconds. Visualisation has to be handled externally. In this case, however, simple text output should be enough. A simple loop could look this this:

```
dReal *spherePosition = dBodyGetPosition(sphereBody);
while (spherePosition[2] > 1)
    {
        printf(Current height: %d\n, spherePosition[2]);
        dWorldStep(world, 0.1);
    }
```

This would output the height of the sphere at the beginning of the simulation, after 0.1 seconds, after 0.2 seconds and so on. Once the sphere hits the floor, the simulation ends. It should be noted that, because this loop has no collision handling whatsoever, the sphere would just fall through the floor an keep falling indefinitely.

1.3 Other Objects in Physics Simulations

Of course, there is more to physics simulations than just spheres and a flat plane. However, detailed understanding of all these things is not necessary in order to read this paper. This section will name and briefly explain the objects that appear in this paper.

There are more geoms than just spheres. Other shapes are boxes, cylinders, cones and capsules (cylinders with a half-sphere placed on each end). All of these shapes are created almost exactly like the sphere in the above example - the only difference is the number of parameters needed to define a geom's size. More complex shapes, called compound colliders, can be built using several other shapes as building blocks. There are also rays that, like planes, do not have a body and thus cannot be moved. The same thing is true for bodies: While ODE uses only one body type, other engines support several body types. A rigid body in Bullet would behave like the body used in the previous section; another type of body is a soft body, which

allows the geom that is associated with it to be deformed and also allows the engine to factor in the object's elasticity during collisions.

The third major component, besides geoms and bodies, are **joints**. Joints are often also called constraints because they constrain the movement of two objects that are attached to that joint. In the real world, the most common type of joint is a hinge joint that attaches a door to a wall. The hinge joint dictates that the door and the wall cannot be moved away from each other, and that the door can only be rotated around one axis. There are several other kinds of joints that all differ in the amount and degree of movements they allow. The human arm is attached to the shoulder via a ball joint, allowing the arm two rotate around the shoulder around two axes; a cone twist joint would restrict the arm's movement to a cone-shaped area. The most generic joint is a joint with six degrees of freedom that allows two bodies to move around each other freely.

A more abstract component that is featured in ODE is a **space**. The most prominent use for spaces is making collision detection more efficient. Imagine, for example, a simulation with two robots running around on a plane. Each robot is modeled using ten boxes that are attached together via joints, allowing the robots to execute basic movements such as walking. This means there are twenty boxes in the simulation. To detect collisions, the engine would have to check for each box if it collides with any other box. In other words, there are more than $2*10^{18}$ possible collisions that the physics engine has to check for. This can be made much easier by creating two spaces, putting all the boxes that the first robot consists of in the first space and all the other boxes in the second space. A space is not tied to a body, and as a result, its shape and position is determined by the geoms that it contains. For easier understanding, the two spaces could be imagined as bubbles that surround the two robots. Now, as long as the robots do not come near each other, we do not need to check for collisions. Only if the two spaces collide (i.e. the two bubbles intersect) is more detailed collision detection even necessary. This makes collision detection a lot easier, especially when several robots instead of just two exist within the simulation.

Of course, even in ODE, there are algorithms in place that make collision detection more efficient even when the user doesn't define spaces. It is also worth noting that spaces offer several other functions - for example, users can declare that the objects in two given spaces cannot collide at all, which is useful for games like tennis where it is save to assume that the two players will never collide. Bullet handles most of this automatically. As a result, there are no user-defined spaces in Bullet.

2 An Introduction to Simspark

2.1 What is Simspark?

Simspark is a generic physical multiagent simulator written in C++[7]. It gives users the ability to write their own simulations by defining all the objects that are part of the desired simulation in a special script language called RSG (Ruby SceneGraph Language).

Simspark is composed of four libraries, called zeitgeist, **oxygen**, kerosin and salt. There are also several plugins. One of those plugins is the RSG parser, which reads RSG files and calls zeitgeist to create all the objects defined in an RSG file.

Zeitgeist itself can be seen as a huge object factory. It can create objects of all classes that it knows of and arrange these objects in a scene graph. In order to run the Hello World example from section 1.2, we would have to register the classes World, Plane and Sphere with zeitgeist and tell it to create one object of each class. Several parameters, like the sphere's size, the floor's plane equation and also their color in the visual representation are all parameters that can be defined in the RSG file.

Oxygen is responsible for running the physics simulation. When zeitgeist wants to cereate a sphere, it calls the respective method in oxygen, and oxygen delegates that call to the physics engine. It is also responsible for collision handling and running the simulation.

Kerosin and salt are not relevant to this paper. Kerosin handles the visualisation of the simulation run by oxygen, and salt provides some helper methods for mathematical operations.

On top of defining objects, users can also define perceptors and effectors. Perceptors are devices that can collect data from the simulation. In the Hello World example, the sphere could be outfitted with a perceptor that sends a signal when the sphere touches the ground. This would allow the sphere to perceive when it has fallen down. Likewise, an effector can be used to manipulate the simulation. A simple effector for our sphere could be a motor that allows it to roll on the ground once it has fallen down. Finally, users can define agents that make use of several perceptors and effectors to execute a certain behaviour within the simulation. Outfitted with these two devices, the sphere would already be a very simple agent.

A much more elaborate example, and the most significant use of Simspark as of today, is the RoboCup 3D Soccer Simulation League[8]. In this league, teams of several robots compete in the game of soccer. Perceptors allow the robots to determine their position on the field, the position of other players and the location of the

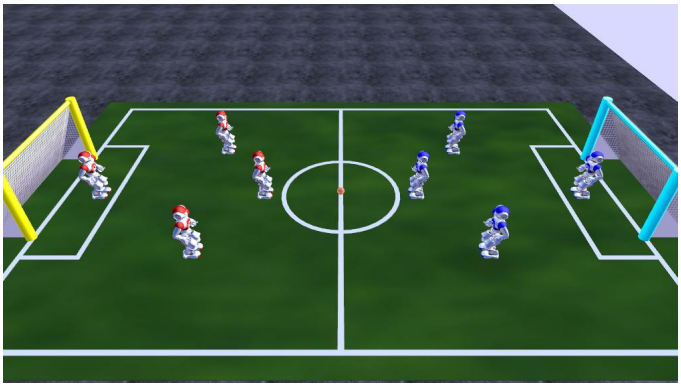

Figure 1: A 3 vs. 3 game in the RoboCup 3D Soccer Simulation League prior to kickoff. Field size is adjusted to accomodate the smaller number of players.

ball. Each of the robot's joints is outfitted with an effector that allows it to move around the field or kick the ball. Computer scientist from all over the world write programs to compute the perceptor's outputs into inputs for the effectors to create the most competent virtual soccer player possible. Their teams then compete against other teams in official competitions.

2.2 What is an Abstract Physics Layer (APL)?

Generally speaking, an APL is a proxy. It declares functions and procedures that are needed to run a physics simulation; however, it doesn't implement them. Instead, it delegates calls to an **implementation** that uses a certain physics engine. The software that wants to run the simulation will only communicate with the APL. More importantly, it does not know which engine is used by the implementation.

This becomes advantageous as soon as support for another physics engine, or even multiple physics engines, is added. If the software communicates directly with the physics engine, changing the physics engine means that the software itself has to be changed, as well. However, if the software uses an APL as a proxy, changing the physics engine means that only the implementation needs to be changed. This is assuming that the implementation's interface remains the same over the course of these changes.

Example: In C++, to create a sphere in ODE, one has to call a method called dCreateSphere that will return a dGeomID. Other parts of the software will use this dGeomID to work with this sphere. However, in Bullet, one has to create an object called a btSphere and Bullet will return a pointer to that object. Of we change a part of a software that created a sphere in ODE so that it now creates a sphere in Bullet, this creates a snowball effect because now, every

other part of the software that used a dGeomID has to use a pointer to an object.

An APL will define a method called CreateSphere and declare that it returns an integer that is used to identify that object. Thus, the software will always receive an integer, no matter which physics engine is used by the implementation. If the implementation uses a different physics engine at one point, it has no effects on the software itself.

2.3 Why an Abstract Physics Layer?

When Simspark was first being developed in 2003, the choice was made to rely on the Open Dynamics Engine (ODE) for the physics simulation. Writing a new physics engine would have been too much work and would have meant reinventing the wheel. ODE was chosen because it was open source, but still on par with professional physics engines in terms of correctness and functionality[1]. However, during the past seven years, the situation has changed.

Development on ODE has been pretty much discontinued, and it has been upstaged by another open-source engine called Bullet. Furthermore, since Simspark is used in official competitions of the Robocup Simulation League, one might consider using a professional physics engine to enhance the simulator's credibility. The main force behind changing the physics engine, however, was a desire for being able to expand Simspark's functionality. Other physics engines that are still in development to this day offer all the functionality that ODE has, and a lot more that ODE does not support. As long as Simspark relies solely on ODE, it can not use the new features that other physics engines have to offer. Furthermore, since most physics engines now are more efficient than ODE, switching the engine would allow users to run more complex simulations without the need for better hardware.

However, simply abandoning ODE and relying on a single other physics engine (e.g. Bullet) was not an attractive option, since it would cause Simspark to cease working on systems on which only ODE is installed. It would also be only a temporary solution, as the engine that replaces ODE could, in the future, be surpassed by yet another physics engine. An APL, however, enables users to choose which engine they want to use and makes it easier to support future physics engines.

2.4 Related Work

An Abstract Physics Layer that offers the possibilities described above already exists. It was created by Adrian Boeing and is called Physical Abstraction Layer (PAL)[3]. PAL currently supports twelve physics engines. Due to the similiraty to what we wanted with an APL for Simspark, PAL has caught much interest in the Simspark community, but was unanimously dismissed for one reason: In supporting so many physics engines, it has had to rely on a common ground between too many components, which means that its functionality is severely limited. Since expanding the functionality of Simspark was the major goal, PAL stands directly against the goals of the APL that we wanted to develop. The decision to not use PAL had already been made by the time I started working on the APL.

The source code of PAL is visible to everyone, so I looked at its documentation[3] to see if I could implement the APL for Simspark in a similar way. However, PAL uses diamond inheritances, and avoiding diamond inheritances was one of the requirements for the APL (see section 3.2 for details), so the same design could not be used for Simspark.

3 Designing the APL

3.1 Simspark's Original Design

The physics simulation in Simspark is handled by a library called oxygen. Oxygen contains an abstract class called PhysicsObject that is derived from BaseNode. All other classes in oxygen are derived from PhysicsObject so that every object created by oxygen can be added to Simspark's scene graph.

Oxygen is designed so that one object created in oxygen encompasses one object in the physics engine. The class *Sphere* in oxygen can create and destroy a sphere in the physics engine, manipulate its attributes and read their values. Zeitgeist ensures that the sphere is properly inserted into the scene graph.

World represents the world in a physics simulation. The world's ID is stored here, and there are methods to manipulate and read the world's parameters, like the strength of gravity.

Space represents the space in a physics simulation using ODE. Since spaces are mostly used for collision detection, collision handling is also done here. Oxygen is able to collide all geoms inside the Space it encompasses and check if it intersects with another space. Collision perceptors are also notified here.

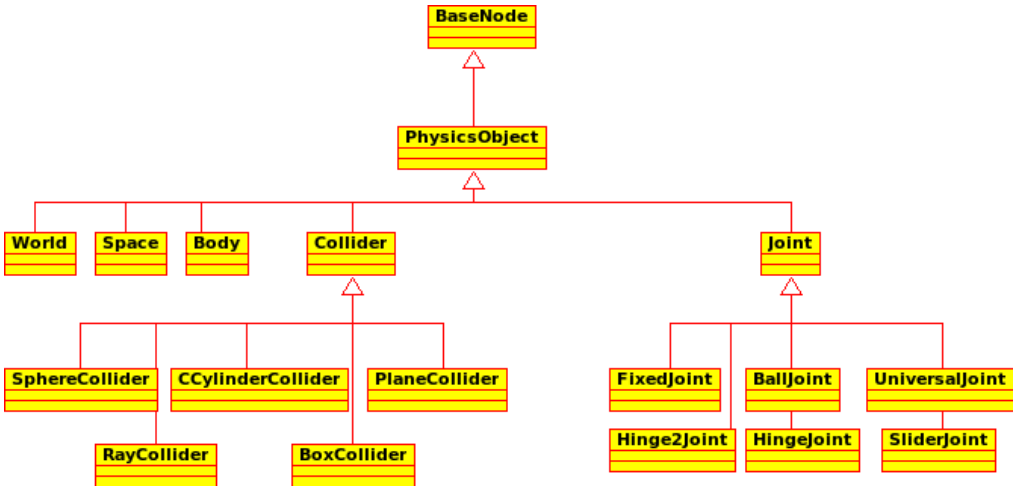

Figure 2: A class diagram showcasing oxygen's original design.

Collider encompasses a geom in the physics simulation. It stores its ID and can work with it via the physics engine's interface. Oxygen originally had only five colliders (capped cylinders, boxes, spheres, rays and planes). Common methods were defined in a superclass called Collider, with CCylinderCollider, BoxCollider, SphereCollider, RayCollider and PlaneCollider being derived from this superclass.

Body encompasses a body, respectively. Since ODE has only one type of body, oxygen originally only had one class called *Body*.

Joint was a superclass to all joint types, similar to Collider being a superclass for all collider types, and works in the same way. Oxygen originally supported six kinds of joints, including hinge joints and ball joints.

3.2 Expanding the Inheritance Tree

In order to expand the functionality, the inheritance tree described in the previous section has been expanded and has also been slightly changed. What oxygen called a Body was actually only one type of several possible physics bodies. In order to cope with this, Body has been renamed to RigidBody and a new class called Body is now an abstract superclass for all types of physics bodies. Three other body types soft bodies, which have elastic properties and can be deformed; static bodies, bodies with no weight that cannot be moved; and dynamic bodies, which are defined but not documented within Bullet have been added to the design.

For colliders, the three shapes that already existed (box, sphere

and capsule) are now derived from a new abstract superclass called ConvexCollider. PlaneCollider and RayCollider are still derived directly from Collider. Two other convex shapes (cones and cylinders) have been added to the design as new child classes of ConvexCollider. Three other collider types (concave, compound, empty) have been added in addition to the new ConvexColliders.

Finally, an abstract superclass called Generic6DOFJoint[3] has been put in front of all other joint classes. Only cone twist joints have been added to the tree, boosting the number of supportable joint types from six to eight.

Space and World remained unaltered.

3.3 Requirements

One of the first requirements was that out of the existing libraries, only oxygen should be changed unless changing code outside of oxygen was absolutely unavoidable. Eventually, four more detailed requirements were raised to ensure that after oxygen's redesign, Simspark would still work with it. Early attempts at development showed that violating these requirements would cause errors within other components of Simspark.

1) An engine-independent class must exist for every kind of physics object.
This requirement emerges naturally due to the nature of an APL. The example in section 2.2 showed that different engines use different methods and different parameters to create the same kind of physics object. Since the libraries outside of the APL do not know which engine is used, the APL must offer an engine-independent class for spheres that can act as a correspondent for the other components. Naturally, an engine-independent class like this must exist for every other physics objects.

2) Engine-independent classes must be derived from BaseNode.
Simspark arranges all objects, including physics objects, in a scene graph. The functionality for this is provided by BaseNode, the class that PhysicsObject is derived from. All classes that inherit from BaseNode can be added to the scene graph. However, if a class is not derived from BaseNode, it is impossible to add objects created by that class to the scene graph.

[3] 6DOF = six degrees of freedom

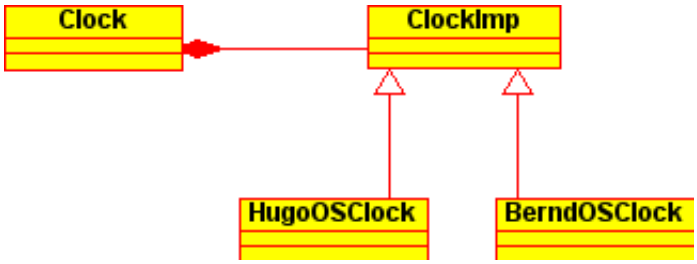

Figure 3: The Bridge Pattern: A small example

3) No diamond inheritances.
A diamond inheritance can form if multiple inheritances are used. If a class D is derived from two superclasses B and C, and both B and C are derived from a class A, then A is an ambigious superclass of D. Diamond inheritances within oxygen mean that BaseNode becomes an ambigious superclass of at least one class in oxygen. Tests showed that this causes Simspark to crash when zeitgeist tries to cast a physics object up to BaseNode. The immediate cause for this error was not found because in the previous example, casting D up to A is possible without any problems using C++. It might have been possible to fix this error by changing the respective code in zeitgeist, but changing code outside of oxygen was to be avoided. Thus, the requirement was raised to avoid diamond inheritances and little time was spent searching for the cause of this crash.

4) All engine-specific classes must implement a common interface.
This requirement was raised to assure the quality of the design. If classes using different physics engines offered different interfaces, the APL could be designed to cope with this, but it would be far from the best solution. If a common interface is enforced, the APL's code can be structured a lot better. It also helps people who want to add support for different physics engines because a well documented interface will tell them what they have to implement.

3.4 The Bridge Pattern

The Bridge Pattern is a design pattern specified by Erich Gamma and the Gang of Four in [4]. Since it had a big influence on this redesign, it shall be explained in this section with a brief example.

Example: There are two operating systems HugoOS and BerndOS. Both operating systems offer a function that returns the current system time in milliseconds. However, the function is called

getSystemTime in HugoOS and retrieveTime in BerndOS. We want our software to work on both operating systems without creating different versions for each system. Using the Bridge Pattern is a possible solution for this problem.

To follow the Bridge Pattern, we need to create four classes. Clock is a platform- independent class that offers a function getTime which returns the current system time in milliseconds. Other parts of the software will make calls to this method if they want to retrieve the system time. The second class, ClockImp, is an interface that declares a method called getTimeImp. Finally, HugoOSClock and BerndOSClock both implement getTimeImp by using the respective method to retrieve the system time from their operating system and returning the result.

At runtime, if HugoOS is used, an object of the class HugoOSClock is created, cast up to ClockImp and a pointer to this object is stored in Clock. Likewise, if BerndOS is used, an object of the class BerndOSClock is created, cast up to ClockImp and a pointer to this object is stored in Clock. Now, other parts of the software can just call Clock::getTime() to retrieve the system time.

3.5 Oxygen's Final Redesign

The final design is a variation on the Bridge Pattern. An excerpt is shown above. Going by this excerpt, it is easy to understand the complete design; however, showing the entire design in UML would take a diagram spanning several pages.

It should be noted that I used the suffix "Int" instead of "Imp" because in the bridge pattern, all classes using "Imp" as a suffix are actually just interfaces. This naming convention is a little more consistent. The other discrepancy is that the classes that should be called ODE* are now called *Imp (which is another reason I couldn't use this suffix for the interfaces). The reason for this alteration is explained in section 4.2.

4 Implementing the Abstract Physics Layer

4.1 Renaming Body and CappedCylinder

As mentioned in section 3.2, the Body class was renamed to RigidBody and a new abstract class called Body was inserted to serve as a superclass for all types of physics bodies. This means that every file that included body.h now includes rigidbody.h instead, every command that created a body now creates a rigid body, and instead

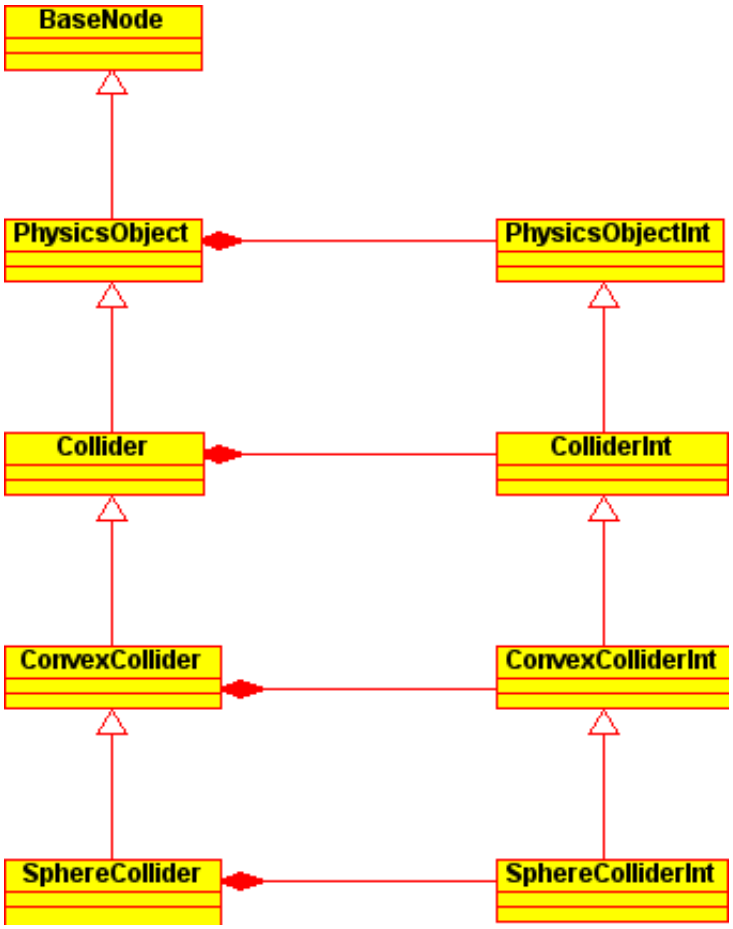

Figure 4: Oxygen's new design (excerpt)

of defining *body* nodes in the RSG files, *rigid body* nodes are now defined there.

CappedCylinder is a deprecated name formerly used by ODE that has been updated to Capsule. Depending on the version of ODE installed, sticking to the name CappedCylinder causes either a compiler warning or, in some cases, even an error. To prevent this from happening, every occurence of the term *CappedCylinder* in Simspark has been changed to *Capsule*. For the sake of consistency, this includes engine-independent classes, RSG files and even comments.

4.2 Applying the Bridge Pattern

In oxygen's original design, there was one class called SphereCollider for spheres. The header file of this class did not include ODE, however; this meant that SphereCollider's interface was engine-independent from the start. It also turned out that no class outside oxygen included ODE (with the exception of one plugin). Now, the goal was to make SphereCollider and all other classes in oxygen engine-unspecific.

In accordance with the Bridge Pattern, I created two new classes called SphereColliderInt (where the Bridge Pattern suggests SphereImp) and SphereColliderImp (where the Bridge Pattern suggests ODE-Sphere). There was not enough time to even start adding Bullet support, so BulletSphere was not added at that point.

Another alteration was made because it allowed engine-switching at runtime. The bridge pattern, in its original form, has to be applied at compile time: If the example in section 3.4 was implemented, preprocessor commands would be necessary in two places: For obvious reasons, HugoOSClock cannot be compiled on BerndOS, and vice versa. Since only one of these classes can ever be compiled at the same time, a simple if-statement is not enough for deciding how the implementation object is instanciated in Clock. We need preprocessor commands in this place as well to avoid getting compiler errors because of an unknown class name.

To support switching the engine at runtime, all engine-specific code was confined to a plugin. The plugin was called "odeimps" - however, all classes within that plugin received neutral names, e.g. SphereImp. Back in oxygen, in the Sphere class, an object of the class SphereImp is created and a pointer to that object is stored. This means that oxygen doesn't know which engine is currently in use. Now, users can compile all the plugins if the corresponding engines are installed. A ruby script that runs every time Simspark is

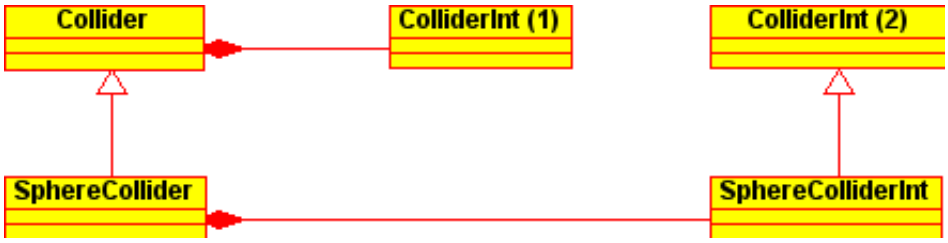

Figure 5: Visualisation of the problem described in section 4.3

launched decides which plugin is used. Changes to this ruby script don't require recompilation of Simspark to be effective. Conclusively, the decision which engine is used is made at runtime.

4.3 Storing IDs

In ODE, every object is given an ID. A function that creates a physics object always returns the ID of the newly created object. This ID is necessary to access the object at a later point. During developement, the question arose where this ID should be stored.

IDs use ODE-specific data types. A geom is identified by a dGeomID, a world is identified by a dWorldID and so on. Due to this circumstance, one might think that the most plausible solution is to store the IDs as a member of the implementation object. After all, each physics object in the scene graph has exactly one implementation object, and each implementation object stores exactly one unique ID. Doing this can lead to serious problems.

SphereCollider is derived from Collider. A Collider owns exactly one ColliderImp, and a SphereCollider owns exactly one SphereColliderImp. Most methods that are implemented in the Collider superclass need the dGeomID, so the original approach was to declare the dGeomID as a member of ColliderImp. SphereColliderImp inherits the attribute from its parent.

Now, we create a Sphere. SphereCollider delegates the call to its implementation object, the implementation creates a sphere within ODE and stores the ID it receives from ODE. Next time, we want to know the collider's xyz coordinates. This method is the same for every collider type, so it is implemented in Collider. The call is delegated to ColliderImp; however, the object this call is delegated to is now a different ColliderImp than the one that the ID was formerly stored in. The ID in this object is still zero and the result is an error.

To solve this, the ID is stored in Collider. After creating a sphere

within ODE, SphereColliderImp returns the ID of the newly created sphere to its owner. SphereCollider stores the ID in its parent. Now, when the next method is called, the ID is present within Collider and passed to ColliderImp as an argument. The call now works as intended.

This matter may seem trivial, but it can be very hard to understand. In fact, it was one of biggest obstacles during development. It is easy to overlook that SphereCollider inherits ColliderImp from its parent. In other words, every SphereCollider owns both a SphereColliderImp as well as a ColliderImp. These are two different objects, so there's no way an ID stored in SphereColliderImp could be present in ColliderImp unless it is passed around.

4.4 Generic Data Types

The above solution creates another problem: If ODE stores its ID in specific data types, e.g. dGeomID, and the abstract layer has to avoid using these data types, how do you store the ID in the abstract layer? The solution is quite simple: An isolated test case quickly revealed that ODE's IDs are, in fact, data of type long and that, more importantly, casting back and forth between these IDs and long was perfectly save and had no visible side effects. This means that after creating a Sphere, SphereColliderImp simply casts the dGeomID to long and returns it to its owner. Next time, SphereColliderImp receives the ID from its owner, casts it back to a dGeomID and does whatever it has to do.

This gets a little more complicated when references to ODE-specific objects are passed around, e.g. a reference to a dContact, denoted as dContact& in C++. However, the solution was quite similar. An empty class called GenericContact was declared to handle this. Now, just like the dGeomID was cast to a long and cast back to a dGeomID, the dContact& can be cast to a GenericContact& and cast back to a dContact& with no visible side effects.

4.5 Static Methods

Some of the predefined methods in oxygen were static. This was a problem during most stages of development because the Bridge Pattern can not be used for static methods. The pointer to the implementation object is a member of the object who owns that implementation object, but members can not be used in static methods. Following the Bridge Pattern to the letter, it is also not possible to declare the implementation's methods as static because, at least

in C++, static methods can only be called if you know which class they are in. Since the Bridge Pattern suggests classes with prefixed names like ODESphereCollider, BulletSphereCollider etc. and we do not know which engine is used when we write the code, the only remaining option is using preprocessor commands every time we want to call a static method. First, I used a complicated workaround to avoid using preprocessor commands in too many places.

This problem fixed itself once we decided to create a plugin. Since the plugin uses neutral class names, e.g. SphereColliderImp, and that class name is the same for every physics engine, we can call methods like SphereColliderImp::StaticMethod() without risk.

4.6 The Joint Class and GetParameter, SetParameter

Joints in ODE have many parameters. To access and manipulate them, ODE offers two methods dJointSetParameter and dJointGetParameter. These methods take an ID as an argument; using an ID table, ODE finds out which parameter the user wants to access. The catch is that there are different types of joints, and these methods are slightly different for each joint type.

When I started working, oxygen used a very tricky design around this. The class Joint featured two methods GetParameter and SetParameter that took an ID, as well as more than twenty methods (one for each parameter) that automatically called these two methods with the right IDs. However, since the implementation of GetParameter and SetParameter is different for each joint type, these methods were not implemented, but declared as pure virtual, in Joint. Usually, this design would lead to pure virtual methods being called, but since Joint was declared as an abstract class, it never actually happened. Instead, Joint's child classes inherited all these methods, implemented GetParameter and SetParameter and the design as a whole worked.

The solution for this seemed simple enough. All the methods in Joint that used GetParameter and SetParameter could remain unaltered; GetParameter and SetParameter could remain pure virtual in Joint. Each subclass of Joint could then use its implementation object in GetParamter and SetParameter. However, doing this caused Simspark to crash at runtime, claiming that a pure virtual method had been called. I went back to this problem several times over the course of development, but could never figure out the true reason for this crash.

The only option that I found was using a workaround and implementing GetParameter and SetParameter in the Joint class. Of

course, Joint just delegates the call to JointImp. Fortunately, ODE offers a method that takes a jointID and returns the type of the specified joint. This method could be used in JointImp's implementation of SetParameter and GetParameter, followed by a switch that triggered the correct behaviour for each Joint type.

4.7 Plugins

There was only one plugin that used ODE directly when I started working. Applying the Bridge Pattern to this plugin would have been a waste of time. Instead, the plugin can just be rewritten to support other physics engines and users can define the version of the plugin they want to use in the ruby file.

5 Conclusion

5.1 Example Scenario

In this scenario, I explain the steps taken by Simspark if someone decided to run the "Hello World"-example explained at the beginning of this paper in Simspark. It is assumed that all the necessary classes are registered with zeitgeist, and that the odeimps plugin is loaded. The user wrote an RSG file that declares a world with earth gravity running along the z axis, a plane with the plane equation *z = 0* and a sphere of radius 1 at the position (0, 0, 50).

After a startup sequence, the RSG Parser plugin starts parsing this RSG file. First, it reads the command declaring the world. Thus, the RSG parser tells zeitgeist to create a world. Zeitgeist is an object factory and it knows of the class *World*, which is defined within oxygen. It creates an object of this class and inserts it into the scene graph. When the object of the class *World* is created, its member *mWorldImp* is automatically instantiated with a new object of the class *WorldImp*. Once zeitgeist calls *World::CreateWorld()*, *World* delegates this call to *WorldImp::CreateWorld()*. *WorldImp* then uses *dWorldCreate()* to create a new world within ODE and casts the return parameter, a *dWorldID*, to an integer. This integer is returned as a return parameter to *oxygen::World*, where it is stored.

Next are the gravity parameters, which are (0, 0, 9,81). Similar to what happened above, *World::SetGravity()* is called with the respective parameters. World calls *WorldImp::SetGravity()* and needs to pass the previously stored ID on to this method as a fourth parameter. Then, *WorldImp* calls *dWorldSetGravity()* within ODE as

one would expect. After that, the creation and setup of the plane and the sphere follow the exact same pattern.

5.2 Summary

Before the APL was implemented, oxygen included ODE's header file and was full of ODE-specific code. To use a different physics engine, one would have had to completely rewrite oxygen. Even after that, oxygen had relied specifically on the new physics engine. Without an abstract physics layer, supporting two different physics engines at the same time with the same version of the software is almost impossible.

With the APL being there, oxygen does not use one specific physics engine anymore. Instead, it delegates calls to the APL, which itself delegates the calls to the desired physics engine. With a typical APL using the Bridge Pattern, it is possible to let the same version of a software use different physics engines, and the decision which engine is used is made at compile time.

Due to the engine-specific code being implemented as a plugin that can be chosen at runtime, it is even possible to decide on a physics engine at runtime. All that is needed to support another physics engine is a plugin that uses the desired physics engine. Since the amount of plugins that can be implemented is not limited, it is possible to support an indefinite number of physics engines.

It should be noted, however, that there is no guarantee that simulations will behave identically with every physics engine. A simulation that uses soft bodies will only work with Bullet and other engines that support them, but not with ODE, because ODE doesn't support soft bodies. It should be noted that, in order to prevent a crash, the ODE plugin should still implement some functionality for soft bodies. It could, for example, create a rigid body instead and generate an error message that advises the user to use the Bullet plugin.

6 References

[1] Koegler, Marcus - Simulation and Visualisation of Agents in 3D Environments - 2003 - University of Koblenz-Landau
[2] Rollmann, Markus - Spark, A Generic Simulator - 2004 - University of Koblenz-Landau
[3] Boeing, Adrian - PAL: Physical Abstraction Layer - 2004-2009
[4] Erich Gamma et. al. - Design Patterns : Elements of Reusable Object-

Oriented Software - 1994 Addison-Wesley
[5] ODE Wiki (http://opende.sourceforge.net/wiki/index.php/Main_Page)
(13.11.2010)
[6]Bullet Development Blog - http://bulletphysics.org/wordpress/
(12.01.2011)
[7]Simspark Wiki - http://Simspark.sourceforge.net/wiki/index.php/Main_Page
(12.01.2011)
[8]http://en.wikipedia.org/wiki/RoboCup_Simulation_League
(12.01.2011)