LocoTest: Deploying and Evaluating Physics-based Locomotion on Multiple Simulation Platforms

Stevie Giovanni and KangKang Yin

National University of Singapore
{stevie,kkyin}@comp.nus.edu.sg

Abstract. In the pursuit of pushing active character control into games, we have deployed a generalized physics-based locomotion control scheme to multiple simulation platforms, including ODE, PhysX, Bullet, and Vortex. We first overview the main characteristics of these physics engines. Then we illustrate the major steps of integrating active character controllers with physics SDKs, together with necessary implementation details. We also evaluate and compare the performance of the locomotion control on different simulation platforms. Note that our work only represents an initial attempt at doing such evaluation, and additional refinement of the methodology and results can still be expected. We release our code online to encourage more follow-up works, as well as more interactions between the research community and the game development community.

Keywords: physics-based animation, simulation, locomotion, motion control

1 Introduction

Developing biped locomotion controllers has been a long-time research interest in the Robotics community [3]. In the Computer Animation community, Hodgins and her colleagues [12, 7] started the many efforts on locomotion control of simulated characters. Among these efforts, the simplest class of locomotion control methods employs realtime feedback laws for robust balance recovery [12, 7, 14, 9, 4]. Optimization-based control methods, on the other hand, are more mathematically involved, but can incorporate more motion objectives automatically [10, 8]. Recently, data-driven approaches have become quite common, where motion capture trajectories are referenced to further improve the naturalness of simulated motions [14, 10, 9]. For a more complete review, We refer the interested readers to the recent state-of-the-art report on character animation using simulated physics [6].

With so many locomotion control methods available now in academia, we cannot help but wonder: Can these methods work beyond their specific dynamics formulations or chosen physics engines? If they do generalize well across different physics engines, to what degree does the selection of a particular engine affect the stability and performance of the character control? What components are involved in porting a control scheme onto different simulation platforms? And how difficult would they be?

We show in this paper that it is relatively easy to deploy one type of locomotion control scheme [4] to multiple simulation platforms. Furthermore, the choice of simulation engines is not crucial, if the controller itself is sufficiently general and robust. We

hope our work can encourage more researchers to demonstrate their character control schemes on publicly-available physics engines, for easy benchmarking and comparison.

From the perspective of game development, an observation is that passive object dynamics and Ragdoll simulation are very common features in action games today; yet the adoption of active character control is rare. As a result, simulation engines are tested extensively for their ability to simulate the motion of passive objects, such as an object knocking down a brick wall; while as far as we know, there is no test on physics engines for their ability to simulate active objects. We hope our work can help increase the awareness of game and engine developers, who usually work on one designated platform with tight schedule and budget constraints, that it is now the time to push more active character control into games and physics engines.

The locomotion control scheme we choose for deployment is the generalized biped walking control method of Coros et al. [4]. This scheme produces simple yet robust walking controllers that do not require captured reference motions, and generalize well across gait parameters, character proportions, motion styles, and walking skills. Moreover, the authors released their code online as a Google project named cartwheel-3d(`http:://code.google.com/p/cartwheel-3d/`). cartwheel-3d is built on top of an open-source dynamics engine ODE (Open Dynamics Engine), which is commonly used in the animation research community for its superior constraint accuracy. We further choose two simulation engines that are more popular in the game development community: PhysX and Bullet. PhysX is supported by NVIDIA and currently dominates the market share among the released game titles. Bullet is open source and has great overall performance. Other popular game physics engines today include Havok and Newton. Due to the space limit, we will not discuss these engines further. The biomechanics and robotics communities, however, are often skeptical of simulation results in the graphics literature because of the use of physics simulators that are also associated with games. Therefore, we choose a fourth engine Vortex to test the locomotion controllers. Vortex is currently the leading dynamics platform in the mechanical engineering and robotics industries. Its simulation tools have been widely used in virtual prototyping and testing, virtual training, mission rehearsal, as well as in serious games.

We will first give an overview of the chosen simulation platforms in Section 2. We then discuss the key steps in integrating character control and simulation with physics SDKs in Section 3. The implementation details are further discussed in Section 4. Readers who are not interested in these details can safely bypass Section 4 and jump directly to the evaluation Section 5. We concentrate only on implementation and performance issues pertinent to the porting of locomotion controllers. For evaluation and comparison of other aspects of physics engines, such as constraint stability or collision detection accuracy, we refer the interested readers to [1], where a series of simple tasks with passive objects and constraints are performed and compared.

2 Simulation Platforms Overview

ODE (Open Dynamics Engine) is an open source library for simulating rigid body dynamics and collisions [13]. The library is mostly written in C++, and provides both C

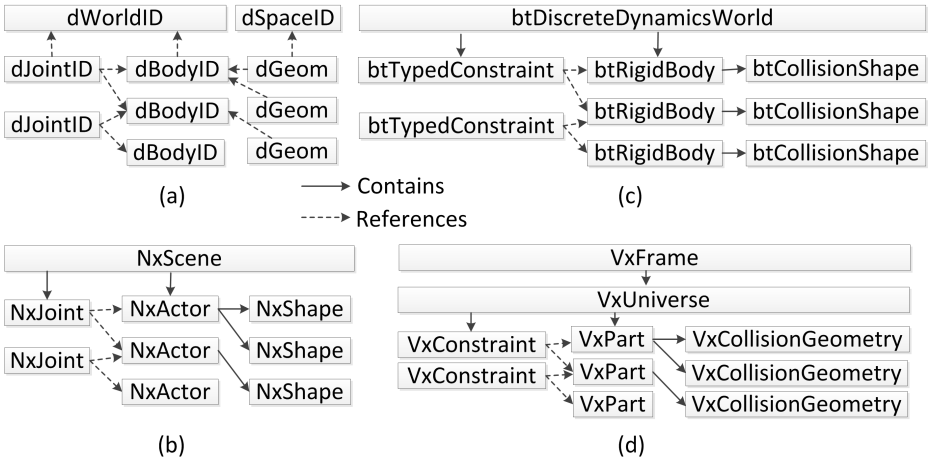

Fig. 1: *Partial architecture diagrams relevant to active rigid character control and simulation of four physics SDKs: (a) ODE (b) PhysX (c) Bullet (d) Vortex.*

and C++ interface. All classes and functions are prefixed with "*d*". ODE uses fixed-time stepping. The latest ODE version 0.11.1 was released in October 2009, and currently ODE's development seems to be suspended.

The NVIDIA PhysX supports rigid and soft body dynamics, collision detection, fluids, particles, vehicle and character controllers [11]. The SDK has a C++ programming interface, and all classes are prefixed with "*Nx*". Fixed and variable time steps are possible. PhysX also provides GPU acceleration for fluids, clothes, and soft bodies, with a CUDA implementation. The latest PhysX SDK V3.0 was released in June 2011, which is a major rewrite of the PhysX SDK. Our code based on V2.8.4 does not work directly on V3.0.

Bullet provides rigid body and soft body dynamics, collision detection, and vehicle and character controllers [5]. It is open source and free for commercial use on many platforms including PLAYSTATION 3, XBox360, Wii, PC, Linux, Mac OSX and iPhone. Bullet is implemented in C++ and all classes are prefixed with "*bt*". It supports both variable and fixed time steps. The latest version of Bullet is 2.78 released in April 2011, which we use in our code.

The CMLabs Vortex supports rigid body dynamics, collision detection, fluids, particles, cables and vehicles [2]. It is a C++ toolkit and all classes are prefixed with "*Vx*". Vortex allows arbitrary changes to the structure of the simulated system in between steps, without any need to reconfigure or restart the simulation. It also integrates with graphics toolkits such as OpenSceneGraph (OSG) and Vega Prime. Fixed and variable time steps are possible. Our code is implemented on Vortex 5.0.1, and the latest release was in August 2011 version 5.0.2.

Fig. 1 shows partial architecture diagrams of the four engines. Relevant classes will be referred to in the following sections. From a programmer's perspective, PhysX and Vortex provide better documentation than ODE and Bullet. Although ODE and Bullet are open source so direct access to the code helps where documentation is lacking. All

Pseudocode 1 : **Character control and simulation pipeline**

```
1: Create a simulation world; //Section 4.1
2: Create objects; //Section 4.2 and 4.4
3: Create joints; //Section 4.3 and 4.4
4: Loop:
5:     Apply control and accumulate forces;
6:     Detect contact and collisions;//Section 4.5
7:     Timestep;
```

dynamics engines provide sample code on basic rigid body simulation. Note that the character controllers provided by PhysX and Bullet do not use "real" physics, but are rather descendants of customized and dedicated code in traditional games to move a player. Such controllers employ heuristic kinematics and quasi-physics to achieve the illusion of controllable and responsive characters. Pure physics-based character controllers are still challenging to construct in games today, and this paper taps the tip of the iceberg by deploying a truly physical locomotion control scheme on various simulation platforms.

3 Character Control and Simulation Pipeline

We deploy the generalized biped walking controller of Coros et al. [4], who released an implementation in C++ on top of ODE. The simulation pipeline for active rigid character control is listed in Pseudocode 1. To switch dynamics engines, each step needs to be mapped to SDK-specific code. We explain Line 5 and Line 7 in this section, and defer all other implementation details to Section 4.

Line 5 of Pseudocode 1 basically is where the controller computes active control torques and accumulates external forces for each rigid link of the character. Then these torques and forces are passed to the physics engine. We refer interested readers to the paper [4] and our source code for details of the control algorithm. Line 7 of Pseudocode 1 steps the dynamics system forward in time for a small time step: constrained equations of motion are solved and kinematic quantities are integrated. Here the major difference of the four engines is whether they separate collision detection and timestepping into two parts. ODE separates them into two distinct steps as shown in Line 6 and 7; while PhysX, Bullet, and Vortex integrate collision detection into their timestepping so Line 6 should really be merged into Line 7 for these three platforms.

ODE provides two timestepping functions: *dWorldStep* and *dWorldQuickStep*. The first one uses the Dantzig algorithm to solve the LCP (Linear Complementarity Problem) formulation of the constrained equations of motion, while *dWorldQuickStep* uses the iterative PGS (Projected Gauss Seidel) method which sacrifices accuracy for speed. Both methods use a fixed user-defined time step. Hereafter we use ODEquick to refer to simulations on ODE using the iterative solver. The PGS method is also used in PhysX and Bullet, and the users can specify the maximum number of iterations allowed. Vortex provides both a standard and an iterative LCP solver, just like ODE. We only test its standard solver *kConstraintSolverStd* in this work. PhysX, Bullet, and Vortex can advance fixed time steps as well as variable time steps during timestepping. For ease of comparison, we always use fixed time steps for all four engines in all our experiments.

	ODE	PhysX	Bullet	Vortex
simulation world	*world* *space*	*scene*	*world*	*frame*
timestepping function	dSpaceCollide(...) dWorldStep(...)	->simulate(...)	->stepSimulation(...)	->step()

Table 1: *ODE decouples the simulation world into a dynamics world and a collision space. Thus it takes two commands to step the system forward. While PhysX, Bullet, and Vortex unify the two concepts and integrate collision detection into the timestepping.*

4 Implementation Details

This section details the major procedures in supporting character control using different SDKs: managing a simulation world; instantiating rigid bodies and joints; and handling contacts and collisions. Commonalities and differences between ODE, PhysX, Bullet, and Vortex are summarized and contrasted. We also illustrate with pseudocode and snippets where necessary.

4.1 Simulation World

A simulation *world* is the virtual counterpart of the physical world that hosts the virtual objects and constraints governed by physics laws in dynamics engines. The control component of cartwheel-3d maintains a world of its own too, containing a copy of all the objects and constraints being simulated. Deploying the control on a new physics engine is thus equivalent to mapping between the SDK's world and the controller's world, as well as passing data between them.

The initialization of the simulation *world* involves specification of: dynamic properties of the world such as gravity; collision properties such as maximum number of contacts and maximum penetration depth allowed; and default contact materials such as coefficient of restitution and friction. ODE decouples the concept of the simulation world into a dynamics world which handles rigid body dynamics and a collision space which handles collision detection, as shown in Table 1. In PhysX, Bullet, and Vortex, however, one world handles both dynamics and collision. The world detects collisions inside the timestepping function shown in Table 1.

One point worth noting is the great flexibility and modularity of Bullet. Many parts of the Bullet world, including the world itself, can be switched to different implementations. We use the default dynamics world *btDiscreteDynamicsWorld*, for which we choose the default broadphase and narrowphase collision detection implementation and the default constraint solver *btSequentialImpulseConstraintSolver*. It is also interesting to note that Bullet actually contains the ODE quickstep solver, due to the open source nature of both engines, so that the users can switch the solvers easily if they so choose.

4.2 Objects

Objects within the simulation world can be classified into three categories: the first type carries the kinematic and dynamic properties including position, velocity, and mass

	body (dynamic object)	shape (geometric object)	static shape (static object)
ODE	*body*	*geom*	*geom* (not attached to a body)
PhysX	*actor*	*shape*	*actor* (with no dynamic properties)
Bullet	*rigidBody*	*collisionShape*	*collisionObject*
Vortex	*part*	*collisionGeometry*	*part* (frozen by users after creation)

Table 2: *Dynamic, geometric, and static objects have different names in different SDKs. We refer them as bodies, shapes, and static shapes in our discussion.*

and inertia; the second type specifies the geometric properties, or shape, of an object. Hereafter we refer to the first type as bodies, and the second type as shapes. Shapes are usually attached to bodies for collision detection, and one body may be associated with one or more collision shapes. Common collision primitives include box, sphere, capsule, and plane. Advanced collision shapes such as height field, convex hull, and triangle mesh are also supported by these four engines. There is a third kind of objects that we call static shapes, which represent static objects such as ground which usually do not move and are only needed for collision detection. Static shapes do not attach to bodies and do not posses time-varying kinematic quantities. Table 2 lists the names of these types of objects in different physics SDKs.

In ODE there is a special type of *geom* called *space*, which can contain other *geoms* and *spaces*. The collision world in ODE is in fact a *space*. In PhysX, a body is called an *actor*. Actors with specified dynamic properties, such as mass and inertia, are dynamic; otherwise they are static. Similarly, a body in Vortex, called a *part*, can be either static or dynamic. A *part* is dynamic by default after creation, but users can freeze it as a static object. ODE, PhysX, and Vortex all support multiple compound collision shapes attached to a single body. But in Bullet, each body or static shape can only have one collision shape attached. Also Bullet does not support relative transformation between collision shapes and the bodies they attach to. Therefore collision shapes in Bullet posses identical position and orientation as the bodies they are associated with. To achieve more sophisticated collision shapes with various initial configuration in Bullet, users can first define a compound shape to store all the needed component shapes, and then encapsulate different relative transformations into the child shapes.

4.3 Joints

Joints are constraints that limit the relative movements of two otherwise independent bodies. We classify the types of joints by counting how many rotational and translational Degrees of Freedom (DoFs) a joint permits. Table 3 lists the common joint types supported by the four SDKs. Again the same type of joint may be named differently on different platforms.

All the engines support a large set of common joint types. PhysX and Bullet also provide a special kind of six-DoF freedom joint. Each DoF of a freedom joint can be locked or unlocked independently, to model almost any type of joints. Snippet 1 illustrates how to construct a universal joint from a freedom joint in PhysX. In addition to the listed joint types, PhysX also implements distance joint, point in plane joint, point on line joint, and Pulley Joint. Bullet provides an additional cone twist constraint which

#DoFs	0	1R	2R	3R	1T	1R1T	6
ODE	fixed	hinge	universal	ball-and-socket	slider	piston	
PhysX	fixed	revolute		spherical	prismatic	cylindrical	freedom
Bullet		hinge	universal	point2point	slider		freedom
Vortex	RPRO	hinge	universal	ball-and-socket	prismatic	cylindrical	

Table 3: *Common joint types supported by the four simulation engines. We classify the types of joints by counting how many rotational and translational DoFs a joint permits. For example, 1R1T means 1 rotational DoF and 1 translational DoF.*

```
NxD6JointDesc d6Desc;
d6Desc.xMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.yMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.zMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.twistMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.swing1Motion = NX_D6JOINT_MOTION_FREE;
d6Desc.swing2Motion = NX_D6JOINT_MOTION_LOCKED;
NxD6Joint * d6Joint=(NxD6Joint*)gScene->createJoint(d6Desc);
```

Snippet 1: *Constructing a universal joint using a freedom joint in PhysX.*

is a special point to point constraint that adds cone and twist axis limits. Vortex supports even more types of constraints, such as the Angular2Position3, CarWheel, and Winch constraint. The zero-DoF joint RPRO can also be relaxed to model a six-DoF joint. Moreover, Vortex supports velocity constraints such as the ScrewJoint, Differential, and GearRatio constraints. In our case of walking control, however, the basic rotational joints of one to three DoFs are already sufficient.

To achieve desired joint behavior, correctly setting up the various joint parameters is the key. For instance, most rotational joints are modeled by an anchor point and up to two rotational axes. A third axis is computed implicitly as a cross product of the two defined axes. These axes form the right-handed local coordinate frame, as well as help define joint limits. In PhysX however, except for the 6DoF freedom joint, all other joints adopt a left-handed local coordinate frame. Understanding the joint local frame is crucial when using the built-in Proportional Derivative (PD) controllers to power or motorize certain joints, where users can specify the desired joint angles and velocities.

Specifying joint angle limits is usually straightforward except for ball-and-socket joints, for which only PhysX supports the specification of the local rotational axes. In PhysX and Bullet, however, limits can be specified for a twist-and-swing decomposition of the angular freedoms of 6-DoF freedom joints. We thus create ball-and-socket joints from freedom joints when using these two SDKs. In ODE and Vortex, we circumvent this problem by attaching an additional angular motor constraint to the two bodies that a ball-and-socket joint constrains, and then specifying limits for the angular motor along its three motor axes as shown in Snippet 2.

Lastly, ODE provides users contact joints for handling collisions. Contact joints prevent two bodies from penetrating one another by only allowing the bodies to have an "outgoing" velocity in the direction of the contact normal. We will discuss this further in Section 4.5 where we look at collision handling mechanisms of the four simulation platforms.

```
dJointID j = dJointCreateBall(worldID, 0); //create ball-and-socket joint
dJointAttach(j, parentBody, childBody);//attach to bodies
...//set joint parameters;
dJointID aMotor = dJointCreateAMotor(worldID, 0); //create angular motor
dJointAttach(aMotor, parentBody, childBody);//attach to bodies
...//set amotor parameters;
dJointSetAMotorAxis (j,0,rel,x,y,z);//set motor axis 0
dJointSetAMotorAxis (j,2,rel,x,y,z);//set motor axis 2
//set joint limits for swing angle 1
dJointSetAMotorParam(aMotor, dParamLoStop, minSwingAngle1);
dJointSetAMotorParam(aMotor, dParamHiStop, maxSwingAngle1);
...//set joint limits for swing angle 2 and twist angle
```

Snippet 2: *Angular motors help specify joint limits for a ball-and-socket joint in ODE.*

```
NxRevoluteJointDesc revoluteDesc; //joint descriptor
revoluteDesc.actor[0] = actor1; //attach actors constrained by the joint
revoluteDesc.actor[1] = actor2;
revoluteDesc.setGlobalAnchor(NxVec3(p.x,p.y,p.z)); //set the anchor point
revoluteDesc.setGlobalAxis(NxVec3(a.x,a.y,a.z)); //set the rotation axis
//create the joint
NxRevoluteJoint * revoluteJoint =
(NxRevoluteJoint *) gScene->createJoint(revoluteDesc);
```

Snippet 3: *Revolute joint setup in PhysX*

4.4 Creating Objects and Joints

Another difference of the four physics SDKs is the way objects and joints are created. In PhysX every object has a descriptor, which is used to specify all the arguments of an object before its creation. Snippet 3 shows a sample on how to create a revolute joint in PhysX. ODE, on the other hand, allows users to specify parameters for objects after their creation. Snippet 4 shows the code for creating the same revolute joint (although called hinge joint) in ODE. Bullet adopts yet another approach by providing object-creating functions with long lists of arguments. Snippet 5 illustrates this point. Vortex supplies two types of procedures, one with many arguments as in Bullet; and the other with less arguments during creation, but users need to specify additional parameters later on as in ODE.

These examples also show that different physics engines add objects into the simulation world in different ways. PhysX starts with a scene (i.e., the simulation world) as the main object. Every other object is created using one of the creation methods of the scene with corresponding descriptors as arguments. In ODE users first apply for an object ID from the world, and then create the object with detailed specifications. In Bullet and Vortex, users add objects to the world after their creation.

4.5 Collision Detection and Processing

All four simulation platforms supply built-in collision detection engines. PhysX, Bullet and Vortex are generally acknowledged for providing more powerful and robust collision systems, such as stable convex hull collision detection and warmstart of contact constraints. From the point of view of walking control of a single character with

```
dJointID j = dJointCreateHinge(worldID, jointGroup); //create the joint
dJointAttach(j,body1,body2); //attach bodies constrained by the joint
dJointSetHingeAnchor(j, p.x, p.y, p.z); //set the anchor point
dJointSetHingeAxis(j, a.x, a.y, a.z); //set the rotation axis
```

Snippet 4: *Hinge joint setup in ODE.*

```
//create the constraint with all necessary parameters
btTypedConstraint * constraint =
  new btHingeConstraint(object_a,object_b,pivot_in_a,pivot_in_b,...);
//add the joint into the world
world->addConstraint(constraint,bool disableCollision=true);
```

Snippet 5: *Hinge constraint setup in Bullet*

rigid links, however, the basic ODE collision detection is already sufficient. The major difference of these platforms is how they activate the collision detection module. PhysX, Bullet, and Vortex provide fully automatic collision detection and seamless integration with the dynamics solver. Users only need to define collision shapes, and then all the collision constraints are generated implicitly and desired collision behavior will automatically happen. In ODE, however, users have to call the broad phase collision detection manually before timestepping, as shown in Table 1. In addition, users need to supply a callback function to further invoke the narrow phase collision detection, as shown in Snippet 6. Lastly, users must explicitly create contact joints from detected collisions for the dynamics solver to take into account the collisions, as shown in the for loop of Snippet 6.

4.6 Collision Filtering

The most effective way that users can influence the simulation performance is by filtering unnecessary collision detections. There are several mechanisms to filter collisions, as listed in Tabel 4.

	object pairs	group pairs	bit mask	callback
ODE		✓	✓32-bit	must
PhysX	✓	✓	✓128-bit	optional
Bullet			✓32-bit	optional
Vortex	✓	✓		optional

Table 4: *Collision filtering mechanisms.*

ODE provides built-in support to filter collisions between different groups of shapes during the broad phase collision detection. Each *geom* also has a 32-bit "category" and 32-bit "collide" bitfield to bypass collision testing between shapes in different categories. As explained earlier, users have to define a collision callback function for ODE to incorporate contact and collision constraints into the dynamics. Inside this callback, users have complete freedom to further filter out collisions between shapes. In the context of character control, collisions between joined limbs, such as the upper arm and the lower arm, should be ignored.

In PhysX, collisions between pairs of jointed bodies are disabled by default. Moreover, users can change the collision detection behavior between an arbitrary pair of shapes or actor groups by setting related flags. Different from the 32-bit bit mask mechanism in ODE and Bullet, users are able to specify a 128-bit group mask for each shape in PhysX. This mask is further combined with user specified constants and operators to generate a boolean value indicating if contacts should be generated for a pair of shapes.

```
void collisionCallBack(dGeomID shape1,dGeomID shape2,...)
{
    //filter collision
    ......
    //call the narrow phase collision detection function
    //cps records contact position, normal, penetration depth etc.
    num = dCollide(shape1,shape2,max,&(cps[0].geom),sizeof(dContact))

    //generate contact joint for each detected collision
    for(int i=0;i<num;i++)
    {
        //define the material parameters for the collision in cps
        ......
        //create a contact joint based on the collision point cps
        dJointID c=dJointCreateContact(worldID,contactGroupID,&cps[i]);
        //assign feedback variable for collision postprocessing
        dJointSetFeedback(c,&(jointFeedback[i]));
    }
}
```

Snippet 6: *ODE narrow phase collision detection callback function*

Bullet does not have built-in support to filter collisions between a given pair of shapes or groups. It does have a bitwise filter mechanism similar to that of ODE that can be used to tailor collision detection between shapes and groups. In Vortex, users can disable collision detection between pairs of objects, assemblies of objects, and groups of objects (identified by IDs), either through the appropriate container classes or through an intersect filter class.

Users can also define customized callback functions for total control over the collision or intersect filtering mechanism, in PhysX, Bullet, and Vortex, just like in ODE. The difference is that callback functions are optional except for ODE. It is worth noting that mask-based collision selection happens a lot further up the tool chain than the callbacks do, so collision masks are preferred to callbacks if they are sufficient for your purpose. PhysX and Vortex can also automatically detect inactive dynamic bodies, bodies that do not move for a period of time, and put them into sleep until external forces wake them up. Sleeping bodies are not detected for collision or simulated to save time.

4.7 Collision Postprocessing

Postprocessing of contact and collision forces is needed when the locomotion controller wishes to regulate the Ground Reaction Forces(GRFs) between the character and the ground, to calculate GRF-based feedback controls, or simply to monitor or visualize the GRFs. Collision postprocessing should be done after the timestepping, when the contact and collision forces or impulses have been resolved by the constrained dynamics solver.

Postprocessing contacts in ODE involves reading back the contact information from the *jointFeedback* array initialized on the last line of Snippet 6. Bullet initializes a collision dispatcher during the creation of the simulation world, and from the dispatcher the contact information can be obtained for postprocessing. In PhysX, users subclass

walk style		inplace	normal	happy	sneak	chicken	drunken	jump	snake	wire
control parameter	desired speed	0.0	1.0	2.0	1.5	2.5	0.5	1.0	3.0	-1.0
	cycle duration	0.6	0.6	0.5	0.6	0.5	0.6	0.5	0.5	0.6
	step width	0.12	0.12	0.1	0.14	0.12	0.2	0.15	0.13	0.1
motion distance	ODEquick-ODE	2.3e-4	1.6e-4	2.8e-4	9.5e-4	1.0e-3	1.6e-3	7.9e-4	6.6e-4	9.1e-4
	PhysX-ODE	5.5e-2	1.9e-2	9.6e-2	8.3e-1	8.4e-2	5.6e-2	9.6e-2	7.3e-2	1.5e-1
	Bullet-ODE	1.7e-2	1.2e-2	1.5e-2	5.9e-2	1.3e-2	1.7e-2	2.9e-2	1.5e-2	1.4e-2
	Vortex-ODE	1.8e-3	3.5e-3	7.1e-3	2.7e-2	6.4e-3	1.9e-3	6.1e-3	1.1e-2	3.7e-3

Table 5: *Motion deviation analysis. The simulated motion on ODE serves as the baseline. ODEquick uses the iterative LCP solver rather than the slower Dantzig algorithm. We investigate nine walking controllers: inplace walk, normal walk, happy walk, cartoony sneak, chicken walk, drunken walk, jump walk, snake walk, and wire walk.*

NxUserContactReport and register an instanced object of this class during the initialization of the simulation world. Then *onContactNotify(...)* of the object receives the contact information for each pair of shapes or actors that has requested contact notification through proper flag setting. Similar to PhysX, Vortex users can subclass *VxIntersectSubscriber* to access contact events before or after timestepping. However, if users just need to read back the contacts after the dynamics has stepped forward, a simpler way is to access *VxDynamicsContact* via a pointer of *VxUniverse*.

5 Performance Evaluation and Comparison

We test nine controllers walking in various styles provided by the original cartwheel-3d online distribution. For common simulation parameters we use the cartwheel-3d defaults on all platforms, e.g., a ground friction of 0.8; a fixed time step of $0.5ms$ etc. Since each controller can walk the character successfully with a range of parameter settings, such as the desired walking speed, duration of one walk cycle, and width of the steps, we manually choose one point in the control parameter space as listed in Table 5. Then we measure the distance between the motion simulated on each engine and that on ODE. That is, we use the motion simulated on ODE as the baseline. The distance $d(m,\tilde{m})$ between two simulated motions m and $\tilde{m}$ is defined as follows:

$$d(m,\tilde{m}) = \frac{\sum_{k=1}^{n}\sum_{i=1}^{l} ||\mathbf{p}_k^i - \tilde{\mathbf{p}}_k^i||}{nlh} \tag{1}$$

where n is the number of frames in $\tilde{m}$ and l is the number of links of the character. We record ten cycles of a simulated walk at $30Hz$ as m and $\tilde{m}$, starting from the fifth cycle when the walk has converged onto its limit cycle. Then m is time aligned with $\tilde{m}$ and resampled to n frames for comparison. $\mathbf{p}_k^i$ is the center of mass location of each limb in the character root frame. h is the height of the character for normalization.

Perceptually the limit cycles of the simulated walks from all nine controllers are quite similar, although there are cases the step length or width is noticeably different. We encourage the readers to check the accompanying demo video yourself. The difference of the beginning start-up cycles is also notable. In fact, due to the initial difference,

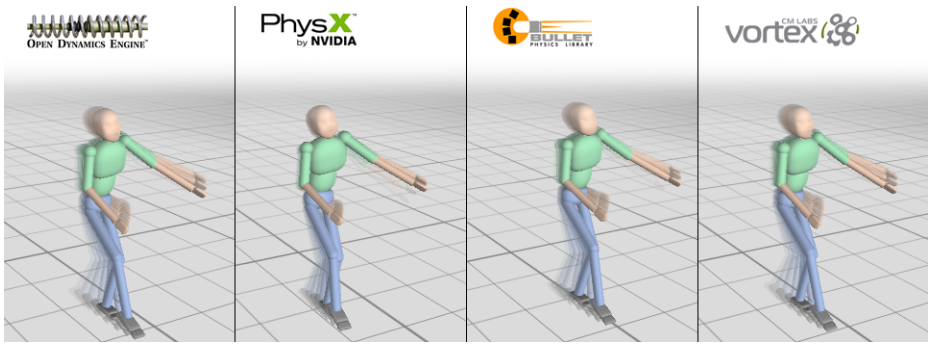

Fig. 2: *Screen captures at the same instants of time of the happy walk, simulated on top of four physics engines: ODE, PhysX, Bullet, and Vortex (from left to right).*

Engine	bound1	bound2	bound3
ODE	18	1.6	1.0
ODEquick	18	1.4	1.0
PhysX	150	0.9	–
Bullet	100	1.3	1.0
Vortex	35	1.6	1.2

Table 6: *Stability analysis with respect to the size of the time step in ms.*

Engine	$d(m_{8/9}, m_{10/11})$	$d(m_{8/9}, m_{14/15})$
ODE	0.185	0.036
ODEquick	0.196	0.037
PhysX	0.172	0.006
Bullet	0.174	0.014
Vortex	0.157	0.002

Table 7: *Stability analysis of the normal walk with respect to external push.*

some characters walk diagonally rather than on the default straight line. Quantitatively, the simulations on ODE and Vortex resemble more, and their averaged limb position difference never exceeds 3% of the character height. Usually the simulations from PhysX differ more. Fig. 2 shows a side-by-side comparison of the same frames of the happy walk simulated on each platform.

We test the stability and robustness of the normal walk controller on each engine with respect to the size of the simulation time step. The original cartwheel-3d uses a default simulation time step of $0.5ms$ on top of ODE. We further search for three time steps as shown in Table 6. bound1 is the largest time step before the simulation becomes unstable; bound2 is the largest time step before the simulated character falls; and bound3 is the largest time step before the distance between the simulated motion from the default motion simulated at the default time step becomes larger than 0.1. Note that in PhysX the character falls using a time step larger than $0.9ms$, when the motion distance is 0.06. We can see that simulations in PhysX and Bullet are much more stable at large time steps than in ODE and Vortex, but they do not differ much in terms of the stability of the walking controller once the simulation moves into the stable region.

We also test the robustness of the normal walk controller on each engine with respect to external perturbations. The character gets pushed by a planar force of $(250,150)N$ backward and sideways for $200ms$ at the onset of the tenth cycle m_{10}. Using the motion from the eighth and ninth cycles, denoted as $m_{8/9}$, as the baseline motion, we then compare how much $m_{10/11}$ and $m_{14/15}$ differ from the baseline $m_{8/9}$. Table 7 shows that the motion error caused by the external push diminishes quickly, and the character

Engine	1 character	10 characters	100 characters	100 characters with 6 threads
ODE	0.099	0.96	9.9	×
ODE-quick	0.064	0.61	6.6	×
PhysX	0.300	1.57	14.2	11.6
Bullet	0.107	0.96	11.0	–
Vortex	0.120	1.60	17.4	12.4

Table 8: *Average wall clock time of one simulation step in milliseconds using the default simulation time step 0.5ms. Timing measured on a Dell Precision Workstation T5500 with Intel Xeon X5680 3.33GHz CPU (6 cores) and 8GB RAM. Multithreading tests with PhysX and Vortex may not be valid due to unknown issues with thread scheduling.*

eventually goes back to the original limit cycle. Note here we only measure the distance between the feet positions, as the controller uses a foot placement strategy to regain balance so the upper body postures do not differ too much after the perturbation.

The computational cost for simulating 10 characters is roughly distributed as follows: 2% on rendering; 18% on control; 65% on simulation including collision detection and constraint solving (except ODEQuick which is faster). Table 8 shows the average timing of one simulation step using the default time step 0.5*ms*, for one character, ten characters, and a hundred characters. We see a near linear degradation of the performance, probably mainly because our characters all walk independently. We also tried simulating multiple characters with GPU acceleration turned on in PhysX, but we did not observe any performance gain with our NVIDIA graphics card GeForce GTX 570. This is because rigid body collisions are still processed by the CPU in the PhysX version we use. Furthermore, we tested the multithreading capability of PhysX and Vortex. Unfortunately our tests with 6 threads on our 6-core machine did not show significant speedup either. This contradicts with released tests from PhysX and Vortex. In diagnosing this problem, we found that only two of our six cores are active no matter how many threads we specify for the engine. This may be caused by the Python interface or wrapper used in our software, or unknown issues in the interaction between the Windows thread scheduler and Python. Bullet also provides multithreading and GPU acceleration, which we have not tested due to lack of documentation.

6 Conclusion

We have deployed the generalized locomotion controller in cartwheel-3d to multiple simulation platforms with ease. Our code is released online at `http://animation.comp.nus.edu.sg/locotest.html`. The porting part of this project was completed within four weeks by a first year graduate student who had no experience in simulation and control but had basic knowledge on computer animation. The original controllers can immediately walk the character successfully after porting, although in slightly different styles. The robustness of the controllers in multiple styles stays across different platforms. Our experience suggests that it is plausible and straightforward to integrate the chosen locomotion controller into game physics today.

We would like to emphasize that our results are specific to the type of controller being tested [4], and its specific implementation in cartwheel-3d. This implementation controls the walking style through explicit PD torques at every simulation time step. The

advantages of such an implementation include: computing the control at each time step is cheap and easy; porting the controller to off-the-shelf physics engines is straightforward; and the simulated character exhibits natural compliance when pushed. The disadvantage is that the simulation has to take smaller time steps, compared to other methods which integrate the equations of motion directly into each control time step. However, all our tested engines provide built-in joint PD controls, some of which use implicit methods to achieve better stability at larger time steps. We plan to explore such stable PD controllers in the near future.

Our performance analysis mainly serves to test the plausibility of deploying the walking control to multiple simulation platforms, and is not for accurately comparing the performance of the physics engines themselves. Different engines have different parameter settings that can trade off among robustness, accuracy, and speed. We use default settings of these parameters on all platforms, which may favor different aspect of the performance depending on the preference of the specific engine. We believe more interactions between the academia and the game industry are needed to achieve better active character control for games and game engines today. Hopefully our effort in this work can serve as a solid starting point.

Acknowledgements: We wish to thank Michiel van de Panne and Stelian Coros for providing helpful suggestions on an early draft of this paper; and Daniel Holz from CMLabs for verifying our implementation details and multithreading test for Vortex.

References

1. Boeing, A., Braunl, T.: Evaluation of real-time physics simulation systems. In: GRAPHITE '07 Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia (2007)
2. CMLabs Simulations, I.: Vortex 5.0.1 vx developer guide (2011)
3. Collins, S., Ruina, A., Tedrake, R., Wisse, M.: Efficient bipedal robots based on passive-dynamic walkers. Science 307(5712), 1082–1085 (2005)
4. Coros, S., Beaudoin, P., van de Panne, M.: Generalized biped walking control. ACM Transctions on Graphics 29(4), Article 130 (2010)
5. Coumans, E.: Bullet 2.76 physics sdk manual (2010)
6. Geijtenbeek, T., Pronost, N., Egges, A., Overmars, M.H.: Interactive character animation using simulated physics. In: Eurographics - State of the Art Reports (2011)
7. Hodgins, J.K., Wooten, W.L., Brogan, D.C., O'Brien, J.F.: Animating human athletics. In: Proceedings of SIGGRAPH 1995. pp. 71–78 (1995)
8. de Lasa, M., Mordatch, I., Hertzmann, A.: Feature-based locomotion controllers. ACM Transctions on Graphics 29(4), Article 131 (2010)
9. Lee, Y., Kim, S., Lee, J.: Data-driven biped control. ACM Transctions on Graphics 29(4), Article 129 (2010)
10. Muico, U., Lee, Y., Popović, J., Popović, Z.: Contact-aware nonlinear control of dynamic characters. ACM Transactions on Graphics 28(3), Article 81 (2009)
11. NVIDIA: Physx sdk 2.8 documentation (2008)
12. Raibert, M.H., Hodgins, J.K.: Animation of dynamic legged locomotion. ACM SIGGRAPH Computer Graphics 25(4), 349–358 (1991)
13. Smith, R.: Open dynamics engine v0.5 user guide (2006)
14. Yin, K., Loken, K., van de Panne, M.: Simbicon: Simple biped locomotion control. ACM Transctions on Graphics 26(3), Article 105 (2007)