Grand Valley State University
ScholarWorks@GVSU

Honors Projects | Undergraduate Research and Creative Practice

Winter 2011

Physically-Based Modeling

Christopher Brinks
Grand Valley State University

Follow this and additional works at: http://scholarworks.gvsu.edu/honorsprojects

Recommended Citation
Brinks, Christopher, "Physically-Based Modeling" (2011). *Honors Projects*. 82.
http://scholarworks.gvsu.edu/honorsprojects/82

This Open Access is brought to you for free and open access by the Undergraduate Research and Creative Practice at ScholarWorks@GVSU. It has been accepted for inclusion in Honors Projects by an authorized administrator of ScholarWorks@GVSU. For more information, please contact scholarworks@gvsu.edu.

Physically-Based Modeling
by Christopher Brinks

Abstract
Physically based modeling is a growing trend in computer animation. There are many implementations available for this topic. The most basic of these involves the movement of single particles (without a shape) moving through space. This implementation involves the movement of particles that have a rigid structure, such as a box or ball, known as rigid bodies. It features a simple box comprised of 8 points moving through space according to the laws of physics as it makes contact with a surface.

Introduction

The movement of particles throughout a 3-dimensional space requires the object's state to be stored. This state, also known as a state vector $\mathbf{Y}(t)$ consists of four parts: position ($x(t)$), a rotation matrix ($R(t)$), linear momentum ($P(t)$), and angular momentum ($L(t)$). With a simple particle, this information alone represents the entire object. A more complicated rigid body consists of multiple points defining the shape, as well as a point representing the object's center of mass. In order to simulate the movement of the rigid body through space, we have to know all of the forces acting on it at any given time t. The change in $\mathbf{Y}$ is represented by the equation:

$$\frac{d}{dt}\mathbf{Y}(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ \omega(t)^* R(t) \\ F(t) \\ \tau(t) \end{pmatrix}$$

From this equation we can keep track of how the object's position, orientation, and the forces exerted on it change over time. To begin the simulation, we create the object (or objects), and give them an initial position, orientation, and define the initial forces acting on it.

Physically-Based Animation

The goal of this project is to provide a simulation that imitates how objects move in the real world. It must therefore follow the laws of physics, especially gravity and momentum. The goal is to define these properties from the real world in the simulation. Once this is accomplished, it is simply a matter of setting up the initial state and letting the animation run itself without any more interference required.

As stated before, this animation features a box moving through a 3-D environment and interacts with a "floor" according to the laws of physics. The box is defined by a list of 24 numbers, where each 3 numbers defines one of the 8 vertices of the box. There are actually two separate lists. One keeps track of the coordinates with the origin at the point (0, 0, 0) in three-space, while the second list is based on the origin being the center of mass of the box. The former is used when determining if a collision has occurred between the object and the floor. The latter is used for animation. The second list is also referenced by another list that defines the faces of the box.

Algorithms

There are four equations that form the backbone of this simulation. They are the equations that form the derivative of $\mathbf{Y}(t)$ mentioned above. The first,

$$x'(t) = v(t)$$

states that the derivative of position with respect to time is the linear velocity of that particle at the given time t.

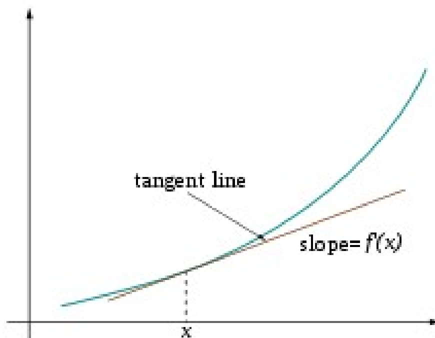

Figure 1. Derivative of position seen as the slope of the tangent line.

Figure 1 shows the slope of a position curve, where *x* represents the independent variable time, and *f(x)* is the dependent variable, position. The slope of the line tangent to *f(x)* is the value of velocity. This of course shows the two-dimensional representation, while the actual implementation in the simulation uses all three dimensions.

The second equation,

$$\boldsymbol{R}'(t) = \omega(t) * \boldsymbol{R}(t)$$

is a simplification of the equation

$$\mathbf{r}'(t) = \omega(t) \times \boldsymbol{r}(t)$$

It represents the derivative of the rotation matrix, which shows the change in the rotation matrix over time.

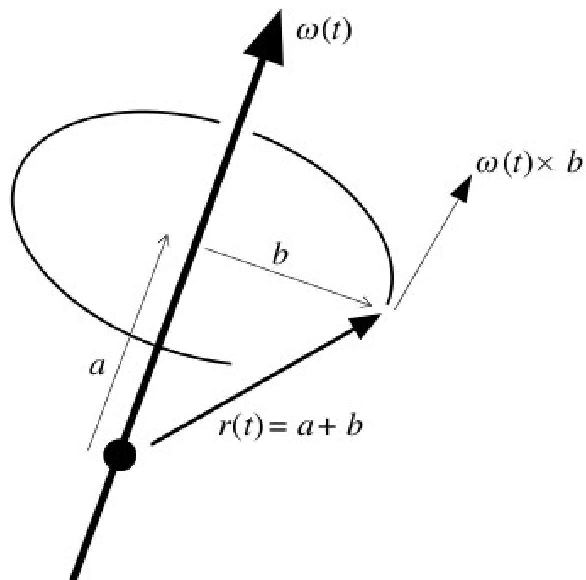

Figure 2. The change in the rotation matrix over time. [2]

This equation can be rewritten as

$$\boldsymbol{r}'(t) = \left(\omega(t) \times \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} \quad \omega(t) \times \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} \quad \omega(t) \times \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \right)$$

Each column of the right side is the direction of the x axis, y axis, and z axis respectively of the rigid body in world space. This is a complicated equation, which can be simplified using a trick. Suppose *a* and *b* are 3-vectors, and $a \times b$ is:

$$\begin{pmatrix} a_y b_z - b_y a_z \\ -a_x b_z + b_x a_z \\ a_x b_y - b_x a_y \end{pmatrix}$$

a^* is defined as

$$\begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix}$$

so

$$a^* b = \begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix} \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_y b_z - b_y a_z \\ -a_x b_z + b_x a_z \\ a_x b_y - b_x a_y \end{pmatrix} = a \times b.$$

Using this information, ***R'(t)*** can finally be written as

$$\boldsymbol{R}'(t) = \omega(t)^* \left(\begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} \quad \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} \quad \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \right)$$

This equation gives us the rate of change of the rotation vector.[2]

The third equation,

$$\boldsymbol{P}'(t) = \boldsymbol{F}(t)$$

is the derivative of the equation

$$\boldsymbol{P}(t) = m\boldsymbol{v}$$

represents the relation that the linear momentum of a body is the sum of the products of the mass and velocity of each particle, in this case, each vertex. Since *m* is constant, we can rewrite the equation for the change in linear momentum over time as

$$\mathbf{v}'(t) = \mathbf{P}'(t) / m$$

or after some substitution as stated earlier,

$$\mathbf{P}'(t) = \mathbf{F}(t).$$ [2]

The final equation involving the state variables is the angular momentum. The equation for angular momentum can be written as

$$\mathbf{L}(t) = r(t) \times \mathbf{P}(t)$$

The angular momentum of the object is based on the linear force being exerted, which creates a torque on the object.

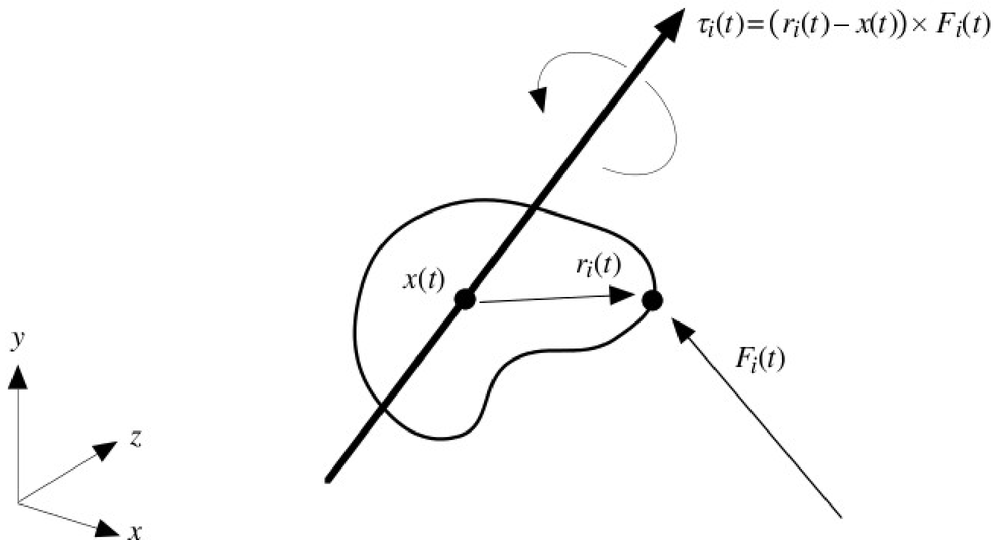

Figure 3. An illustration of the torque as a result of a force acting on an object.[2]

The change in linear momentum over time is equal to the torque being exerted on the object, represented by the equation

$$\mathbf{L}'(t) = \tau.$$

Implementation

The implementation of this project is quite complex. The most basic construct is the rigid body itself. This is implemented as a C++ class. Its data members consist of the state variables (**Y**), the derived quantities of **v** (linear velocity), **ω** (angular velocity), and I^{-1}(inertia tensor inverse), a boolean variable indicating whether or not the object is stationary, the two computed quantities force and torque, and the two lists of vectors described in the previous section. The class also features functions that set up these state variables, and a function designed to make GLUT function calls to animate the object.

The state variables of the rigid body class are stored as either a custom matrix class or custom vector class. The matrix class keeps track of data using a single dimension array of doubles, while the vector class implements an STL vector. Both contain constructors that can either use a default size or accept a

value to use as the size. They also contain some necessary mathematical functions required for the calculations, such as multiplication, division, addition, and subtraction. These are available for situations only if the operation is defined, such as a square matrix multiplied by another square matrix, an addition of a matrix and a vector with the appropriate dimensions, and so on. Also, the matrix class contains definitions for transpose and star operations. The vector contains definitions for cross product and dot product. Almost all of these operations are done through the use of operator overloading, to make it as readable as possible.

The driving force of the simulation is the structure of GLUT programs. The GLUT timer function will animate the object by default every 20 milliseconds, and it will make a call to another function to increment the time by a default value of $1/30^{th}$ of a second. These values are defined as global constants so they can be easily changed. Upon each step, the program will then copy all the relevant state information for each rigid body into STL vectors. At this time, the derivative of the state of each rigid body is calculated using a 2nd order derivative (first an Euler step and then a midpoint evaluation) for a reasonably accurate update of its state according to all of the forces acting on it. Next the program will copy the vector back into the rigid body's state variables.

Once the state has been properly updated, the program will then check for object collisions. If inter-penetration of rigid bodies has occurred, an STL vector is created to keep track of the closest vertices of the box to the floor. These will be used to determine what the next step should be set to. When there is penetration, the program must calculate the best estimate of the time at which this penetration occurred. Once the time is calculated within a reasonable threshold, the lowest vertices are then updated in accordance with the forces that will be exerted on them by the floor in response to the collision. Since this is a basic simulation, these collisions will not take into account any loss of force due to friction or loss of momentum from the slight compression of the objects that would occur in the real world. Execution then resumes from the calculated time of impact by adding the step to the current time.

It should be noted that the visual rendering of the rigid body is updated at a constant interval, whereas the calculations are compute by adding the next step to the current time. Still, in the event of collisions as mentioned in the previous paragraph, the time may not be updated by the size of the step every time. This is done by design to provide the smoothest viewing experience possible.

Conclusion

The end result of this simulation is a box that begins falling and bounces off the floor according to the laws of physics. It seems like a trivial implementation, but this program lays a strong foundation for far more complex animations. The applications of this type of program are limitless, and there are similar, albeit far more robust, programs used as physics engines for nearly every modern animation featured prominently in video games.

References

1. Wikipedia contributors. "Derivative." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 29 Apr. 2011. Web. 1 May. 2011. <http://en.wikipedia.org/w/index.php?title=Derivative&oldid=426589705>
2. Witkin, Andrew and David Baraff. "Physically Based Modeling: Principles and Practice." *Online SIGGRAPH '97 Course Notes.* 1997. Web. January 10, 2011. <http://www.cs.cmu.edu/~baraff/sigcourse/>