CS 480/680: GAME ENGINE PROGRAMMING

PHYSICS

2/14/2013
Santiago Ontañón
santi@cs.drexel.edu
Outline

• Student Presentations
• Basics on Rigid Body Dynamics
• Linear Dynamics
• Angular Dynamics
• Collision Handling
• Additional Features
• The Physics Module
Outline

- Student Presentations
- Basics on Rigid Body Dynamics
- Linear Dynamics
- Angular Dynamics
- Collision Handling
- Additional Features
- The Physics Module
Student Presentations

- Steven Bauer:
  - “Real-time Shadows on Complex Objects”

- Christopher Miller:
  - “Real-Time Strategy Network Protocol”

- Joseph Muoio:
  - “Search-based Procedural Content Generation”

- James Rodgers
  - “Terrain Analysis in an RTS – The Hidden Giant”

- Justin Weaver:
  - “Coping with Friction in Dynamic Simulations”
Outline

• Student Presentations
• Basics on Rigid Body Dynamics
• Linear Dynamics
• Angular Dynamics
• Collision Handling
• Additional Features
• The Physics Module
Game Engine Architecture

- Game Specific
- Game Engine Functionalities
- Resource Management
- Utility Layer
- Platform Independence Layer
- SDKs
- OS
- DRIVERS
- HARDWARE

Dependencies
Game Engine Architecture

- Rendering Engine
- Scripting
- Artificial Intelligence
- Online Multiplayer
- Gameplay Foundations (Game Loop)
- Animation Engine
- Physics
- Audio Subsystem
- Collisions
- Profiling & Debugging
- Gameplay Foundations (Game Loop)
Rigid Body Dynamics

• What do I mean by having a “physics module”?  
  • Basically: mechanics (and in particular, “dynamics”)
  • Typically: rigid body dynamics
  • Some games include other aspects like fluid dynamics, but that is a topic on its own.

• Rigid Body Dynamics:
  • **Classic Newtonian physics**: objects in the simulation will obey Newton’s three laws of motion.
  • **Rigid Bodies**: objects in the simulation cannot be deformed (i.e. constant shape)
Examples

• Lunar Lander (1979)
  • http://www.youtube.com/watch?v=IzdxjaVm_HQ
Examples

- The Incredible Machine (1993)
  - [http://www.youtube.com/watch?v=EJbEDIDDVVc](http://www.youtube.com/watch?v=EJbEDIDDVVc)
Examples

- Angry Birds (2009)
  - http://www.youtube.com/watch?v=9-hjAY0XpvE
Examples

  - http://www.youtube.com/watch?v=ILaxCkPKyJs
Examples

• From Dust (2011)
  • [http://www.youtube.com/watch?v=rqsu0vNv_w0](http://www.youtube.com/watch?v=rqsu0vNv_w0)
Physics Module

- Physics module maintains an internal list of game objects (those that are subject to physics):
  - Maintains physics-relevant information: shapes, mass, velocities, accelerations, etc.

- That list of objects might be separate from the ones maintained by the rest of the game engine.

- Might be shared with collision detection, or might be a complete separate one.
Rigid Body Dynamics Basics

• Units:
  - The physics module needs to measure things (mass, distances, speeds, etc.)
  - You need to set a unit system, common ones:
    • Metric: meters, grams, seconds (standard in physics)
    • MKS: meters, kilograms, seconds (standard in game physics)
    • If you feel very British you can go with the imperial system or US customary units (miles, pounds, etc.) 😊

• Degrees of Freedom:
  - A rigid body has:
    • 3 degrees of freedom in 2D (2 linear and one angular)
    • 6 degrees of freedom in 3D (3 linear and three angular)
  - Interestingly, it is possible to decouple computations for linear and angular motions.
Rigid Body Dynamics Basics

• Units:
  • The physics module needs to measure things (mass, distances, speeds, etc.)
  • You need to set a unit system, common ones:
    • **Metric**: meters, grams, seconds (standard in physics)
    • **MKS**: meters, kilograms, seconds (standard in game physics)
    • If you feel very British you can go with the imperial system or US customary units

• Degrees of Freedom:
  • A rigid body has:
    • 3 degrees of freedom in 2D (2 linear and one angular)
    • 6 degrees of freedom in 3D (3 linear and three angular)
  • Interestingly, it is possible to decouple computations for linear and angular motions.

We will deal with **“linear dynamics”** and **“angular dynamics”** separately.

For circular/spherical shapes, angular dynamics can (sometimes) be ignored.
Rigid Body Properties

Center of Mass (CM)
For **linear dynamics** we can consider that any rigid body is just a point with mass.

Center of mass is the centroid. For each shape supported by our physics engine (circles, boxes, compound objects, etc.), we need a function to compute the center of mass.
Rigid Body Properties

- Shape
- Position (of the CM)
- Mass
- Linear velocity
- Linear acceleration
- Angular velocity
- Angular acceleration
Compound Objects

Each individual part has its CM
Each individual part has its CM

The whole compound object will have its own global CM

\[ \vec{r}_{CM} = \frac{\sum_i m_i \vec{r}_i}{\sum_i m_i} \]
Outline

- Student Presentations
- Basics on Rigid Body Dynamics
- **Linear Dynamics**
- Angular Dynamics
- Collision Handling
- Additional Features
- The Physics Module
Linear Dynamics

• Motion of objects without considering their rotation

• State of an object completely determined by the position of its center of mass (for compound objects, you need the position of all the centers of mass).

• We only care about:
  • Position of center of mass
  • Speed of the center of mass
  • Acceleration of the center of mass
Linear Dynamics

• Basic movement equations:
  
  • Position: \( \overrightarrow{r} = \overrightarrow{r}_0 + \overrightarrow{v}t \)
  
  • Speed: \( \overrightarrow{v} = \overrightarrow{v}_0 + \overrightarrow{a}t \)
  
  • Acceleration: \( \overrightarrow{a} = \frac{\overrightarrow{F}}{m} \)
Linear Dynamics

• Basic movement equations:

  • Position: \( \vec{r} = \vec{r}_0 + \vec{v}t \)

  • Speed: \( \vec{v} = \vec{v}_0 + \vec{a}t \)

Solving these equations, we obtain the typical:

\[
\vec{r} = \vec{r}_0 + \vec{v}_0 t + \frac{1}{2} \vec{a}t^2
\]
Linear Dynamics

• Basic movement equations:
  
  • Position:  \[ \vec{r} = \vec{r}_0 + \vec{v}_0 t + \frac{1}{2} \vec{a} t^2 \]
  
  • Speed:  \[ \vec{v} = \vec{v}_0 + \vec{a} t \]
  
  • Acceleration:

However, this equation is rather useless for physics simulation, since it assumes we know the acceleration (i.e. the forces) that a body will experience over the simulation time. Since we don’t, we need to use numerical integration methods.
Numerical Integration

• Solving the previous equations in a time-stepped way.

• We define a time step $\Delta t$

• **Given**: position, velocity, force at time $t_1$

• **Find**: position, velocity at time $t_2 = t_1 + \Delta t$
Method 1: Explicit Euler

• If we remember that: \[ \vec{v}(t) = \dot{\vec{r}}(t) \]

• Then, we can approximate:

\[ \vec{r}(t_2) = \vec{r}(t_1) + \vec{v}(t_1) \Delta t \]

• And then, we can do the same for the acceleration:

\[ \vec{v}(t_2) = \vec{v}(t_1) + \frac{\vec{F}(t_1)}{m} \Delta t \]
Method 2: Verlet Integration

• Explicit Euler tends to accumulate error, and is unstable. Most games use “Verlet Integration”:

  • Position:
    \[
    \vec{r}(t_2) = 2\vec{r}(t_1) - \vec{r}(t - \Delta t) + \frac{\vec{F}(t_1)}{m} \Delta t^2
    \]

  • Velocity:
    \[
    \vec{v}(t_2) = \frac{\vec{r}(t_2) - \vec{r}(t_1)}{\Delta t}
    \]
What does this mean in code?

- Java example for Explicit Euler:

```java
public void cycleExplicitEulerObject(PhysicsObject po, double timeStep) {
    Vector2d r = po.shape.position;
    Vector2d v = po.linear_speed;

    // next position:
    {
        Vector2d tmp = new Vector2d(po.linear_speed);
        tmp.scale(timeStep);
        r.add(tmp);
    }

    // next speed:
    {
        Vector2d tmp = new Vector2d(po.force);
        tmp.scale(timeStep);
        tmp.scale(1/po.mass);
        v.add(tmp);
    }
}
```

Mathematical equations:

\[
\vec{r}(t_2) = \vec{r}(t_1) + \vec{v}(t_1) \Delta t
\]

\[
\vec{v}(t_2) = \vec{v}(t_1) + \frac{\vec{F}(t_1)}{m} \Delta t
\]
What does this mean in code?

- Java example for Explicit Euler:

```java
public void cycleExplicitEulerObject(PhysicsObject po, double timeStep) {
    Vector2d r = po.shape.position;
    Vector2d v = po.linear_speed;

    // next position:
    {
        Vector2d tmp = new Vector2d(po.linear_speed);
        tmp.scale(timeStep);
        r.add(tmp);
    }

    // next speed:
    {
        Vector2d tmp = new Vector2d(po.force);
        tmp.scale(timeStep);
        tmp.scale(1/po.mass);
        v.add(tmp);
    }
}
```

The code assumes that, before calling this function, all the forces that apply to this object have been added up into `po.force`

\[
\mathbf{r}(t_2) = \mathbf{r}(t_1) + \mathbf{v}(t_1) \Delta t
\]

\[
\mathbf{v}(t_2) = \mathbf{v}(t_1) + \frac{\mathbf{F}(t_1)}{m} \Delta t
\]
Outline

- Student Presentations
- Basics on Rigid Body Dynamics
- Linear Dynamics
- **Angular Dynamics**
- Collision Handling
- Additional Features
- The Physics Module
Angular Dynamics

- Simulation of object rotations

- When not under the effect of any force, rigid solids rotate around their center of mass:
  - This means we can simulate rotation separate from linear dynamics, and obtain a complete description of the body’s motion.

- 2D: easy (almost the same as linear dynamics)
- 3D: complex (3 different degrees of freedom)
2D Angular Dynamics

• Single degree of freedom: \( \theta(t) \)

• Angular speed:

\[
\theta(t) = \theta(t_0) + \omega(t)
\]

• Angular acceleration:

\[
\omega(t) = \omega(t_0) + \alpha(t)
\]
2D Angular Dynamics

- Forces in Rigid Solids

How does this force affect linear and angular acceleration?
2D Angular Dynamics

- Forces in Rigid Solids

Since the force vector crosses the center of mass, this force produces purely a linear acceleration.
2D Angular Dynamics

• Forces in Rigid Solids

How about now?
2D Angular Dynamics

- Forces in Rigid Solids: torque

\[ N = (\vec{p} - \vec{r}) \times \vec{F} \]
2D Angular Dynamics

• Forces in Rigid Solids: torque

\[ N = (\vec{p} - \vec{r}) \times \vec{F} \]

\[ N = |\vec{p} - \vec{r}| |\vec{F}| \sin(\alpha) \]
2D Angular Dynamics

- Now, in linear dynamics, once we know the force, we just divide it by the mass, and we get the acceleration. How about in angular dynamics?

- The equivalent of mass is the **moment of inertia**

- General formula:

\[
I = \int_V \rho(r)r^2 dV
\]
2D Angular Dynamics

- Now, in linear dynamics, once we know the force, we just divide it by the mass, and we get the acceleration. How about in angular dynamics?

- The equivalent of mass is the **moment of inertia**

- General formula:

\[
I = \int_V \rho(r) r^2 dV
\]

Mass is the resistance of an object to change linear velocity.

Moment of inertia is the resistance of an object to change angular velocity around a particular axis.
2D Angular Dynamics

Now, in linear dynamics, once we know the force, we just divide it by the mass, and we get the acceleration. How about in angular dynamics?

The equivalent of mass is the moment of inertia.

General formula:

$$I = \int_V \rho(r) r^2 dV$$

In general, this is complex. So, since the physics module will just support a fixed set of shapes, you can have predefined formulas for the moment of inertia of each shape.
2D Angular Dynamics

- Moment of inertia for common shapes:
  - Circle: \( I = mr^2 \)  
    \[ r = \text{radius} \]
  - Box: \( I = m\frac{h^2 + w^2}{12} \)  
    \[ w = \text{width}, h = \text{height} \]
  - Compound shape: \( I = \sum_{i=1}^{N} m_i r_i^2 \)  
    \[ m_i: \text{mass of part } i, \quad r_i: \text{distance from center of mass of part } i, \text{ to center of mass of compound shape} \]

Explicit Euler for 2D Angular Dynamics

• Angular position:

\[ \theta(t_2) = \theta(t_1) + \omega(t_1) \Delta t \]

• Angular velocity:

\[ \omega(t_2) = \omega(t_1) + \frac{N(t_1)}{I} \Delta t \]
Algorithm, for each simulation step, do (in this order):

1. Add all the forces of the object

\[ \vec{r}(t_2) = \vec{r}(t_1) + \vec{v}(t_1) \Delta t \]
\[ \vec{v}(t_2) = \vec{v}(t_1) + \frac{\vec{F}(t_1)}{m} \Delta t \]
\[ \theta(t_2) = \theta(t_1) + \omega(t_1) \Delta t \]
\[ N = |\vec{p} - \vec{r}| |\vec{F}| \sin(\alpha) \]
\[ \omega(t_2) = \omega(t_1) + \frac{N(t_1)}{I} \Delta t \]
3D Angular Dynamics

- Slightly more complex because:
  - Moment of inertia different for each different axis of rotation
  - It cannot be just computed as a single number
  - Moment of inertia in 3D: a 3x3 matrix called the **inertia tensor**

\[
I = \begin{bmatrix}
I_{xx} & I_{xy} & I_{xz} \\
I_{yx} & I_{yy} & I_{yz} \\
I_{zx} & I_{zy} & I_{zz}
\end{bmatrix}
\]
3D Angular Dynamics

• Slightly more complex because:
  • Moment of inertia different for each different axis of rotation
  • It cannot be just computed as a single number
  • Moment of inertia in 3D: a 3x3 matrix called the inertia tensor

\[ I = \begin{bmatrix}
I_{xx} & I_{xy} & I_{xz} \\
I_{yx} & I_{yy} & I_{yz} \\
I_{zx} & I_{zy} & I_{zz}
\end{bmatrix} \]
3D Angular Dynamics

- Slightly more complex because:
  - Moment of inertia different for each different axis of rotation
  - It cannot be just computed as a single number
  - Moment of inertia in 3D: a 3x3 matrix called the inertia tensor

\[
I = \begin{bmatrix}
I_{xx} & I_{xy} & I_{xz} \\
I_{yx} & I_{yy} & I_{yz} \\
I_{zx} & I_{zy} & I_{zz}
\end{bmatrix}
\]

Elements not in the diagonal do not have an intuitive meaning. They are needed for realistic simulations, but in game engines they are typically set to 0 (since they are 0 for symmetric shapes), and the effect is already realistic enough.
3D Angular Dynamics

• Rotation in 3 dimensions:
  • Can be represented as a 4x4 matrix (recall graphics lecture)
  • As a vector + rotation
  • As a quaternion (the most convenient for 3D physics simulation)

• Quaternion: 4D vector representing a rotation of \( \alpha \) degrees around a vector \( u \):

\[
q = \begin{bmatrix}
 u_x \sin \left( \frac{\alpha}{2} \right) & u_y \sin \left( \frac{\alpha}{2} \right) & u_z \sin \left( \frac{\alpha}{2} \right) & \cos \left( \frac{\alpha}{2} \right)
\end{bmatrix}
\]
3D Angular Dynamics

- Angular velocity represented as a 3D vector:
  \[
  \omega(t) = [u_x \theta(t) \ u_y \theta(t) \ u_z \theta(t)]
  \]

- Angular momentum (3D vector):
  \[
  L(t) = I \omega(t)
  \]
Explicit Euler for 3D Angular Dynamics

- In 3D we need an extra step (update the momentum):

\[ L(t_2) = L(t_1) + N(t_1) \Delta t \]

- Then update the angular speed:

\[ \omega(t_2) = I^{-1} L(t_2) \]
\[
\omega'(t_2) = \begin{bmatrix} \omega_x & \omega_y & \omega_z & 0 \end{bmatrix}
\]

- Then update the angular position:

\[ q(t_2) = q(t_1) + \frac{1}{2} \omega'(t_1) q(t_1) \Delta t \]
Explicit Euler for 3D Angular Dynamics

• In 3D we need an extra step (update the momentum):

\[ L(t_2) = L(t_1) + N(t_1) \Delta t \]

• Then update the angular speed:

\[ \omega(t_2) = \omega(t_1) + \frac{1}{2} \omega'(t_1)q(t_1) \Delta t \]

• Then update the angular position:

\[ l(t_2) = l(t_1) + N(t_1) \Delta t \]

Quaternions need to be renormalized every once in a while, to prevent unexpected effects due to the accumulation of errors.
Outline

• Student Presentations
• Basics on Rigid Body Dynamics
• Linear Dynamics
• Angular Dynamics
• Collision Handling
• Additional Features
• The Physics Module
One-to-One Collisions

• One-to-one collisions can be solved by using the energy preservation law:
  • The energy of the objects before and after the collision must be preserved.
  • Some energy is wasted producing sound and heat, so we will use a preservation constant \( 0 \leq \epsilon \leq 1 \)

• We will assume a very simple collision model:
  • Instantaneous impulse (Newtonian physics)
Impulse

• Impulse: results in an immediate change of velocity
• Represents the change in momentum of an object
• Both objects experience identical impulse but opposite direction

\[ \hat{p} = m \Delta v \]
Impulse

• Impulse: results in an immediate change of velocity
• Represents the change in momentum of an object
• Both objects experience identical impulse but opposite direction

\[ \hat{p} = m \Delta v \]

We need the collision detection module to return the axis of the collision
One-to-One Collisions

- After solving the equations, we obtain the following (that can be directly implemented in code):

\[ \hat{p} = \frac{(\epsilon + 1)(\vec{v}_2 n - \vec{v}_1 n)}{\frac{1}{m_1} + \frac{1}{m_2}} \hat{n} \]

- Where \( \hat{n} \) is a unit vector in the axis of collision (provided by the collision detection module)

- Then, for each of the two objects, update their velocities as:

\[ \vec{v}' = \vec{v} + \frac{\hat{p}}{m_1} \]
Many-to-Many Collisions

- The one-to-one case is solved through the following equations:

\[
p'_{1} = p_{1} + \hat{p}
\]

\[
p'_{2} = p_{2} - \hat{p}
\]

\[
\epsilon \left( \frac{m_{1}v_{1}^2}{2} + \frac{m_{2}v_{2}^2}{2} \right) = \frac{m_{1}v'_{1}^2}{2} + \frac{m_{2}v'_{2}^2}{2}
\]
Many-to-Many Collisions

- If we have three objects where 2 and 3 collide with 1, we have to solve:

\[ \begin{align*}
    p'_1 &= p_1 + \hat{p}_a + \hat{p}_b \\
    p'_2 &= p_2 - \hat{p}_a \\
    p'_3 &= p_3 - \hat{p}_b \\
\end{align*} \]

\[ \epsilon \left( \frac{m_1 v_1'^2}{2} + \frac{m_2 v_2'^2}{2} + \frac{m_3 v_3'^2}{2} \right) = \frac{m_1 v_1'^2}{2} + \frac{m_2 v_2'^2}{2} + \frac{m_3 v_3'^2}{2} \]
Many-to-Many Collisions

- In the general case of $n$ objects and $m$ collisions, you will have an equation system with $m$ unknowns.

- Any standard equation system solving method should do
Angular Collisions

• Wait! Everything we’ve done so far is only for linear collisions. But collisions also affect torque and rotation!

• Luckily, in the same way as for linear dynamics and angular dynamics, we can add angular collision responses on top of linear collision responses.

\[
\begin{align*}
\vec{v}' &= \vec{v} + \frac{\hat{p}}{m_1} \\
\omega' &= \omega + \frac{(\vec{p} - \vec{r})\hat{p}}{I}
\end{align*}
\]

linear \hspace{1cm} \text{angular}
Angular Collisions

- Wait! Everything we’ve done so far is only for linear collisions. But collisions also affect torque and rotation!

- Luckily, in the same way as for linear dynamics and angular dynamics, we can add angular collision responses on top of linear responses.

\[
\vec{v}' = \vec{v} + \frac{\hat{p}}{m_1}
\]

linear

\[
\omega' = \omega + \frac{(\hat{p} - \hat{r})\hat{p}}{I}
\]

angular

We need to:
- Add these equations to the equation system,
- In the preservation of energy equation, consider the velocity of the point of contact (linear + angular)
Angular Collisions

- Wait! Everything we’ve done so far is only for linear collisions. But collisions also affect torque and rotation!

- Luckily, in the same way as for linear dynamics and angular dynamics, we can add angular collision responses on top of linear responses.

\[ \dot{\vec{v}} = \vec{v} + \frac{\hat{p}}{m_1} \]

linear

\[ \omega' = \omega + \frac{(\vec{p} - \vec{r})\hat{p}}{I} \]

angular

Also, notice this is all 2D. Linear collisions are identical for 3D, but for angular collision, we need to consider the inertia tensor, instead of the scalar inertia.
Outline

- Student Presentations
- Basics on Rigid Body Dynamics
- Linear Dynamics
- Angular Dynamics
- Collision Handling
- Additional Features
- The Physics Module
Optimizations

• Islands:
  • It’s is very unlikely that objects that are far away will interact
  • They can be run in separate simulations
  • Divide the space: e.g. octtrees

• Sleep:
  • When objects come to rest, stop simulating them
  • It’s hard to detect when an object has come to rest though
Constraints

• To construct more complex objects such as ragdolls:
  • Hinges
  • Springs
  • Prismatic constraints
  • Wheel
  • Pulley

• They restrict the movement of objects:
  • After the physics simulation step, all the objects are moved back to a position where they satisfy their constraint (constraint solver)
Now, Let’s See This Again

- Angry Birds (2009): Doesn’t it look more impressive now?
  - http://www.youtube.com/watch?v=9-hjAY0XpvE
Outline

• Student Presentations
• Basics on Rigid Body Dynamics
• Linear Dynamics
• Angular Dynamics
• Collision Handling
• Additional Features
• The Physics Module
Your Physics Module

• You will need:
  • Constructor: initializes an empty “physics world”
  • Set gravity
  • Add objects:
    • Shape, position, rotation, mass, moment of inertia, linear and angular velocities
  • Remove objects
  • Simulate one step:
    • 1) Add up all the forces that act on each object
    • 2) Run numerical integration (e.g. Explicit Euler)
    • 3) Resolve collisions
    • 4) Resolve constraints
In your Projects

• Make sure your collision detection modules can:
  • Boolean collision tests
  • Axis of collision (axis along with the collision took place)

• Interesting projects:
  • 2D simulations with multiple shapes (circles, boxes, triangles) and multiple collisions
  • 3D simulations with multiple shapes (since 3D is more complex, I won’t expect multiple collision handling, one-to-one is enough)
  • Handling compound shapes requires constraints, which is complex. I’m not expecting any project to reach this point, but would definitively be very interesting!
Links to Interesting Game Videos

• Scribblenauts:
  • http://www.youtube.com/watch?v=j3HXgvl8lp0
• Cloudberry Kingdom:
  • https://www.youtube.com/watch?v=7qUmIT-GY-M
• Skrillex Quest:
  • https://www.youtube.com/watch?v=17N8wE27wW0
• Proteus:
  • http://www.visitproteus.com/
• Local Indie games:
  • Fractal: https://www.youtube.com/watch?v=ZzrJrmcItMU
  • Jamestown: https://www.youtube.com/watch?v=MH5U92K0JgM
Remember that today:

• Second Project Deliverable:
  • Updated document from Deliverable 1:
    • Address feedback that you got from Deliverable 1
    • Any potential topic change
    • Small description of how the game loop integrates with your demo/game engine
  • Source code:
    • Do NOT send code as an attachment. Send a URL to your code, or to a code repository (SVN, GIT, etc.)

• Submission procedure:
  • Email to (copy both):
    • Santiago Ontañón santi@cs.drexel.edu
    • Stephen Lombardi sal64@drexel.edu
  • Subject: CS480-680 Project Deliverable 2 Group #