06 Game Physics

Tvorba a dizajn počítačových hier (FMFI)
Návrh a vývoj počítačových hier (FIIT)

Michal Ferko

28. 10. 2021

Realistic modeling of the world

- We are mimicking the real world with realistic rendering, environments and animations
- Pre-defined physics animations (exported from a 3D modeling software such as 3ds max) can add physics into our world
- Usually we need to **dynamically** create these animations
- We need to react to user input
- The simplest game physics model: Pong

Pong physics model

- We have a ball moving at constant speed
- We render one frame every 20ms -> 50 FPS
- Whenever the ball hits a paddle, we compute new **velocity**
  - That is, direction and speed
- Paddles are moved only by user input, not affected by the ball

- 3D points: $P, Q, R \ldots \in \mathbb{E}^3$
- 3D vectors: $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \ldots \in \mathbb{R}^3$

Pong physics model (2)

- n = current frame, $n + 1$ = next frame
- Update the position of the ball every frame
  - $P_{n+1} = P_n + (\boldsymbol{v} * \Delta t)$
- Bounce off of the top and bottom of the screen
- Detect if a collision occurred between frame n and frame $n + 1$
  - Is the line $\overline{P_n P_{n+1}}$ intersecting the paddle line?
- On collision, change the ball's velocity
  - Compute the exact point at which the ball will bounce off
  - Change velocity by reflecting the velocity vector
- End game if the ball intersects the left or right of the screen

Complex physics models

- To perfectly simulate the real world, we would need lots of different physical mechanics
- Rigid bodies and their dynamics
- Soft bodies
- Fluids
- Vehicle physics
- Ragdoll physics
- Cloth simulation
- …

Complex physics models - Reality

- Most of these mechanics are very complex and have little usage in games
- All the games out there with physics use **rigid body dynamics**
- Racing and simulator games also include vehicle physics
- Ragdoll physics is used for death animations of characters
- Soft bodies are used rarely, too complex or not well plugged into a game
  - Added value to the game is usually negligible
- Cloth simulation
- Fluid dynamics is usually overkill, animating oceans or water is done using simpler tricks
  - These tricks do not allow the player to interact much with it
  - Portal 2 looks like it’s using fluid dynamics, but it’s a simple trick as well
  - Terraria, Minecraft use extremely simplified models

Rigid body dynamics

- We will focus only on this part of realistic physics
- Our objects will be **rigid bodies** – non-deformable solid objects
- We want to compute the **dynamics** of rigid bodies
  - The movement and rotation of rigid bodies
  - Based on Newton's laws of motion (a.k.a. dynamics)
- There are also several constraints we can place on objects
  - Static objects – not movable by any force
  - Kinematic objects – moved by the users (such as the player character), but not moved by other objects such as grenades or bullets that hit the character
    - Apply force on other objects, any force applied to them is ignored
  - Joints, connected components (car wheel)…
- We will first go through **linear dynamics**

Moving with constant acceleration

- Movement of the object through the world is a *position function* $X(t)$
- We do not know the values of $X(t)$ until all player input up until time t is known
- Constant velocity => $X(t) = X_0 + t * \boldsymbol{v}_0$
- Derivative of the position function is *velocity*
- $\frac{\partial X}{\partial t} X(t) = \boldsymbol{v}(t)$
- 2nd order derivative: *acceleration*
  - When the acceleration is a zero vector, the velocity is a constant vector
  - Otherwise, velocity is a function of time (const. accel.): $\boldsymbol{v}(t) = \boldsymbol{v}_0 + t * \boldsymbol{a}$
  - Or with variable acceleration: $\boldsymbol{v}(t) = \boldsymbol{v}_0 + t * \boldsymbol{a}(t)$

Example

- Parabolic path of a projectile
  - “ballistic shot”
  - Works for all kinds if we ignore air friction
    - Rocks, cannonballs, bullets
- We have an initial velocity $\boldsymbol{v}_0$ and initial position X_0
- Acceleration is equal to gravity, which is a constant
  - $\boldsymbol{a} = (0, -g, 0), g = 9.8ms^{-2}$
- $X(t) = X_0 + t * \boldsymbol{v}_0 + \frac{1}{2}t^2 * \boldsymbol{a}$
- $\boldsymbol{v}(t) = \boldsymbol{v}_0 + t * \boldsymbol{a}$
- $\boldsymbol{a}(t) = \boldsymbol{a}$

Forces

- We want a way to compute the acceleration
- Acceleration is in a relationship with **force**
- $\boldsymbol{f} = m * \boldsymbol{a}$
  - $\boldsymbol{f}$ – force $[N, kg * m/s^2]$
  - m – mass $[kg]$
  - $\boldsymbol{a}$ – acceleration $[m/s^2]$
- Multiple forces affect a single object, we need to compute the position, velocity and acceleration in the next time step with respect to all those forces
- Gravity – we assume the world is flat
- Air friction
- Contact force

Getting the desired positions

- Forces are 3D vectors (direction + magnitude)
- Adding all forces affecting an object in frame N gives us the combined force that will determine the actual movement of the object
- Computing the result
  1. Apply forces $\boldsymbol{f} = \boldsymbol{f}_0 + \boldsymbol{f}_1 + \cdots + \boldsymbol{f}_k$
  2. Compute acceleration from forces $\boldsymbol{a} = \frac{\boldsymbol{f}}{m}$
  3. Integrate acceleration to get velocity $\boldsymbol{v}(t) = \int \boldsymbol{a}(t)\, dt$
  4. Integrate velocity to get position $X(t) = \int \boldsymbol{v}(t)\, dt$
  5. Move objects to the desired position $X(t)$

Linear momentum

- We have a relationship between acceleration and velocity:
  - $\boldsymbol{a} = \frac{d\boldsymbol{v}}{dt}$
- The quantity $\boldsymbol{p} = m * \boldsymbol{v}$ is the **linear momentum** and is related to a force $\boldsymbol{f}$
  - $\boldsymbol{f} = m\boldsymbol{a} = m\frac{d\boldsymbol{v}}{dt} = \frac{d\boldsymbol{p}}{dt}$
- *"The tendency of an object to remain in its current linear motion"*
- If the external force of a system is zero, $\boldsymbol{p}$ is constant
  - Very important for handling collisions

Moving with variable acceleration

- Analytic solution for computing the position and velocity is hard in general
  - We work with forces, so we start from acceleration
  - We will not find the exact equation for $\boldsymbol{v}(t)$ and thus not for $X(t)$
- Impluse forces are simple
- Problems occur with variable forces that are applied continuously
  - Friction, springs, joints…
- Numerical integration solves these equations
  - Advanced topic, not covered here
  - See more in book references

Rotational dynamics

- We have not covered rotation of objects yet!

Position X	$\Rightarrow$	orientation $\boldsymbol{q}$ (a quaternion)
Velocity $\boldsymbol{v}$	$\Rightarrow$	angular velocity ω
Force $\boldsymbol{f}$	$\Rightarrow$	torque τ
Linear momentum $\boldsymbol{p}$	$\Rightarrow$	angular momentum $\boldsymbol{L}$
Mass m	$\Rightarrow$	inertia tensor $\boldsymbol{J}$

- Center of mass important for computing the center of rotation
- The process of computing rotational dynamics is similar to linear dynamics
  - A little more complex, no time to go into detail
- Applying any force to an object is then split into two forces
  - Translational & rotational

Object-object interaction

- So far, we have talked about a single object affected by forces
- What happens when there are more objects?
  - At the moment, objects will pass through each other
- We need to detect when two objects are colliding
- This area of research is called **collision detection**
- Collision response is the next important part
  - We know two objects are colliding, what now?
  - Objects in contact apply forces to each other

Intersections

- Objects (rigid bodies) are just triangular meshes or simple well-defined shapes (spheres, capsules, boxes…)
- We might use other objects to determine intersections
  - For instance rays when shooting from a gun (or picking an object in 3D)
- Other simple helper objects
  - Planes, Spheres, Capsules, Axis-Aligned Boxes, Oriented Boxes…
- When working with intersections, we might want 2 types of result
  - Simple Boolean telling us if we are intersecting or not (our current focus)
  - The actual intersection (point, triangle, mesh, …)

Determining the intersection

- We have lots of different objects
  - Rays, Triangles, Planes, Spheres, Capsules, Axis-Aligned Boxes, Oriented Boxes…
- To determine intersection between either two, a special algorithm has to be used that is designed especially for the two types
- Very easy and fast-to-compute intersection algorithms
  - Ray-Sphere, Ray-Plane, Ray-AAB, Ray-Triangle
  - Sphere-Sphere, Sphere-Plane, Sphere-Triangle
  - AAB-AAB, AAB-Plane, …
  - Fast enough to performs thousands of these per second

Slower intersection algorithms

- Convex meshes
  - Intersections with convex meshes are hard in general
  - Usually, we need to test every single triangle
- Non-convex meshes
  - Even harder than convex meshes
  - Might need to split non-convex meshes into several convex parts for performance reasons
- So slow that only tens to hundreds of tests can be executed per second
  - Heavily depends on mesh complexity
- Mesh-Mesh intersection
  - Might end up testing every triangle-triangle pair $\Rightarrow O(n^2)$
  - Two meshes with 10000+ triangles (not so much for rendering) will take terribly long

Optimizing intersections

- Use simpler shapes
  - Encapsulate complex objects with simpler ones
- No need for exact precision
- Take advantage of hierarchical space-partitioning
- Use bounding volumes

Bounding volumes

- Bound complex mesh geometry with simple objects
- Bounding Spheres
- Axis-Aligned Bounding Boxes (AABB)
- Oriented Bounding Boxes (OBB)
- Use hierarchies of bounding objects for compound objects

(Ray)casting

- Casting an object (ray, sphere…) against a scene determines which objects are hit
  - And in what order
- Used to determine closest objects along a line
  - Can be used to determine all objects we hit
- We can work with the contact points

- Example: shooting a gun in Counter-Strike
  - Cast a ray from the camera in the viewing direction
  - Determine what we hit first – it's instant (usually called "hit-scan")
  - If it's a character ⇒ deal damage
  - If it's a wall ⇒ create a decal
  - Much more robust than fast moving projectiles

Space partitioning

- If we have thousands of objects in the scene and need to compute object-object intersection for each
- Again, $O(n^2)$ tests
- Create a data structure that allows fast querying of possibly intersecting objects
- Discard distant objects fast
- Possibilities
  - Regular grid – BAD
  - Irregular grid – better
  - Bounding Volume Hierarchies (BVH) – much better
  - BSP or kD-trees - best

Space partitioning (2)

- Need to consider dynamic objects
- Our space partitioning system must allow fast rebuilding of the hierarchy
- Or small adjustments in the tree structure during update
- Worst case – rebuilding the whole tree

Collision response

- Once we know two objects are colliding (will be colliding in the next frame), we want to produce a correct response
- Need to take into account physical materials
  - Throwing a ball on concrete vs. throwing a ball on soil
- Collision response is produced as additional forces
- A new force is produced for both objects
- For this, we need to determine the actual intersection locations

Precision problems

- To be fast enough, we cheat, but create several problems that can occur
- Fixed time updates should occur in infinitely short intervals
  - Otherwise, a fast-moving object might pass through walls, because we didn’t catch the moment when it was colliding
- Using static intersections instead of dynamic
  - Using algorithms that consider linear and angular velocity of objects might solve the previous problem
  - However, these algorithms are usually much slower to compute
  - Viable (fast enough) for simple bounding volumes such as spheres
- Solving collision response pair-wise
  - If we solve the collision for A, B and decide to move B so that it collides with C
  - Then, the same happens for B, C => A; C, A => B…

Solutions

- Ostrich approach
  - Ignore the problems and let’s hope a bug will never occur
- Selective approach
  - Select dynamic collision algorithms for more important and fast-moving objects
  - Grenades, bullets, …
  - Any object it will collide with is going to have to use the dynamic intersection version
  - Or at least the “half-dynamic” version (one object dynamic, the other static)

- Know physics limitations
  - Thickness of colliders & maximum object speed
  - Amount of objects
  - Complexity of objects – colliders, rigidbodies, joints…

Unity’s physics engine

- The Transform component of a game objects contains the position X and orientation q
- We add physics through components
- **Rigidbody** - Provides *mass* for our object and allows forces to affect it
  - Other settings such as *drag, collision detection settings, kinematic settings…*
- **Colliders** - Give our objects *shape*
  - Allow us to detect *collision events*
  - Can be triggers that do not actually collide but provide *trigger events*
    - “regions” we can activate
  - Are used to calculate the inertia tensor automatically
- **Joints** and **cloth** - Constraints and non-rigid simulation
- **Character controller** – unaffected by forces, responds to collisions
  - Special component for controllable characters – physically unrealistic
- NVIDIA PhysX (3D), Box2D (2D), Havok (experimental 3D replacement)

Unity’s physics engine scripting

- Physics computation happens after `FixedUpdate`
- The Rigidbody component allows for manipulation of objects with forces
  - Hitting objects with colliders produces contact force
  - Computes center of mass automatically
  - `AddForce, AddTorque, AddForceAtPosition`
  - Manually setting velocity – possible but not recommended
  - Can have physical material (friction and bounciness)
- Collider components
  - Can have density (for Rigidbody auto-mass computation)
  - Can be triggers
  - Provide collision callbacks
    - `OnCollisionEnter, Stay, Exit`
    - `OnTriggerEnter, Stay, Exit`

Execution Order of Event functions

- What Unity events get called in what order
- Most important image for every Unity programmer!!!

https://docs.unity3d.com/Manual/ExecutionOrder.html

Unity’s physics engine scripting (2)

- Physics class
  - Global physics settings (As well as Project Settings => Physics)
  - Overlap tests
  - Cast tests
  - Closest point computation

References

- http://www.essentialmath.com/tutorial.htm
- Mathematics for 3D Game Programming and Computer Graphics
- Essential Mathematics for Games and Interactive Applications: A Programmer’s Guide