PhysX and Vehicles

PhysX

- Collision and Dynamics SDK (now) owned by Nvidia
- Included in UnrealEngine
- Includes a powerful vehicle driving model, which you should use
  - We used to allow rigid body physics libraries but required building driving model from scratch
  - Getting a good driving model is by far the biggest challenge for teams, and was often holding back game quality significantly

Installing PhysX

- Get SDK from https://github.com/NVIDIAGameWorks/PhysX
  - Latest version is 4.1 – make sure you use the right documentation
  - You don't need to register for an account to get this
  - You DO need to register for an account to get PvD (visual debugger, quite useful)
- SDK comes as source – clone or download from GitHub
- Follow Quick Start instructions on GitHub page
  - You need Visual Studio, CMake, and Python 2.7 installed
  - You will end up with a VS solution containing projects for all the PhysX libraries and all the samples
  - A couple of sample projects will fail if you don't have DirectX SDK installed – doesn't matter
- We're going to concentrate on the SnippetVehicle4W sample

Basics of PhysX Vehicle SDK

- Essential tasks
  - Set up library
  - Create world and floor plane
  - Create some meshes
  - Allocate simulation data
  - Allocate actor and add to world
  - Per frame : Set up inputs to drive and steering
  - Per frame : Wheel raycasts
  - Per frame : Tick simulation

Initialization

- PxPhysics
  - Base context for all operations
  - Initialize visual debugger here if you want it

```
// Foundation.
gFoundation = PxCreateFoundation(PX_PHYSICS_VERSION, gAllocator,
      gErrorCallback);

// PvD initialization - optional.
gPvd = PxCreatePvd(*gFoundation);
PxPvdTransport* transport = PxDefaultPvdSocketTransportCreate(PVD_HOST, 5425,
      10);
gPvd->connect(*transport,PxPvdInstrumentationFlag::eALL);

// Physics.
bool recordMemoryAllocations = true;
gPhysics = PxCreatePhysics(PX_PHYSICS_VERSION, *gFoundation,
      PxTolerancesScale(), recordMemoryAllocations, gPvd);
```

Initialization

- PxCooking
  - Utility class for creating meshes in physics
  - Converts a simple vertex stream representing a mesh into an internal structure which it can use for its own purposes
  - PhysX vehicles use meshes for all objects in vehicle system

```
PxCookingParams params(scale);
params.meshWeldTolerance = 0.001f;
params.meshPreprocessParams =
            PxMeshPreprocessingFlags(PxMeshPreprocessingFlag::eWELD_VERTICES |
            PxMeshPreprocessingFlag::eREMOVE_UNREFERENCED_VERTICES |
            PxMeshPreprocessingFlag::eREMOVE_DUPLICATED_TRIANGLES);

mCooking = PxCreateCooking(PX_PHYSICS_VERSION, *mFoundation, params);
```

Initialization

- PxScene
  - Container for all objects in the simulation
  - Global world properties (i.e. gravity)

```
PxSceneDesc sceneDesc(gPhysics->getTolerancesScale());
sceneDesc.gravity = PxVec3(0.0f, -9.81f, 0.0f);

PxU32 numWorkers = 1; // number of threads PhysX can use.
// 0 means PhysX update runs on the main thread.
// 1+ means PhysX update runs on other threads.
gDispatcher = PxDefaultCpuDispatcherCreate(numWorkers);
sceneDesc.cpuDispatcher = gDispatcher;

// The scene filter shader is a function you write which determines whether
// two 'colliding' objects should count as a collision (which you'll get
// notified about) or not.
sceneDesc.filterShader = VehicleFilterShader;

gScene = gPhysics->createScene(sceneDesc);
```

The ground

- Without this your cars will fall forever ;)
- Creating and adding simple objects with standard shapes is very straightforward
- **createDrivablePlane** is a helper function which creates and returns a **PxRigidStatic** object
- **PxRigidStatic** is the base class for rigid objects which aren't expected to move. There are helper functions to create planes, spheres, boxes, etc.
- Note the 'material'. The parameters for the material include the coefficient of restitution. The material properties for two colliding shapes affect how PhysX will handle the collision.

```
//Create a plane to drive on.
// gMaterial - static and dynamic friction, and restitution
gMaterial = gPhysics->createMaterial(0.5f, 0.5f, 0.6f);
PxFilterData groundPlaneSimFilterData(COLLISION_FLAG_GROUND,
      COLLISION_FLAG_GROUND_AGAINST, 0, 0);
gGroundPlane = createDrivablePlane(groundPlaneSimFilterData, gMaterial,
      gPhysics);
gScene->addActor(*gGroundPlane);
```

Set up vehicle support

```
PxInitVehicleSDK(*gPhysics);
PxVehicleSetBasisVectors(PxVec3(0,1,0), PxVec3(0,0,1));
PxVehicleSetUpdateMode(PxVehicleUpdateMode::eVELOCITY_CHANGE);
```

... that was suspiciously easy.

Mesh creation

For a convex mesh, the mesh cooker takes a stream of **PxVec3** objects – the vert positions for each triangle in the mesh.
It produces a **PxConvexMesh** object which is PhysX's internal representation of a mesh optimized for its use.
Created meshes have a hard limit of 256 verts, so the more input verts there are, the more approximate the output mesh will be. If you want more complicated collision, cook a triangle mesh instead.

```
PxConvexMeshDesc convexDesc;
convexDesc.points.count = numVerts;
convexDesc.points.stride = sizeof(PxVec3);
convexDesc.points.data   = verts;
convexDesc.flags = PxConvexFlag::eCOMPUTE_CONVEX;
// This flag asks the cooker to calculate the mesh from the vert list, expecting
// three verts per triangle. Without this flag you must also fill in data to
// describe the actual polys and vert indices of the mesh.

PxConvexMesh* convexMesh = NULL; // this will hold the created mesh
PxDefaultMemoryOutputStream buf;
if(cooking.cookConvexMesh(convexDesc, buf))
{
    PxDefaultMemoryInputData id(buf.getData(), buf.getSize());
    convexMesh = physics.createConvexMesh(id);
}
```

Vehicle description

- The function **initVehicleDesc()** creates a **VehicleDesc** object (this is a convenience class in the sample, not a PhysX class) containing lots of parameters defining the size and performance of the vehicle.
- Play with these numbers to get different behaviour.
- The car in this example has six wheels, of which four are powered and the other two unpowered.
- The function **createVehicle4W()** is the starting point for creating the vehicle model and its simulation data, based off the **VehicleDesc** structure.

The actor for the vehicle

- **createVehicle4W** found in SnippetVehicle4WCreate.cpp starts by making a mesh for a wheel and a mesh for the chassis.
- You will probably replace these later on.
- A vehicle has a **PxRigidDynamic** object for its actor.
- Note difference from ground plane (**PxRigidStatic**) – the scene knows that **PxRigidDynamic** actors will be moving around.

```
PxVehicleChassisData rigidBodyData;
rigidBodyData.mMOI = vehicle4WDesc.chassisMOI;
rigidBodyData.mMass = vehicle4WDesc.chassisMass;
rigidBodyData.mCMOffset = vehicle4WDesc.chassisCMOffset;

veh4WActor = createVehicleActor(rigidBodyData, wheelMaterials, wheelConvexMeshes,
     numWheels, wheelSimFilterData, chassisMaterials, chassisConvexMeshes, 1,
     chassisSimFilterData, *physics);
```

(**createVehicleActor** is a helper function in the sample code which actually makes a **PxRigidDynamic** object and adds the chassis and wheels to it as sub-shapes.)

Wheels setup

- The sample calculates initial local poses for the wheels, i.e. where they are relative to the chassis. You'll need to supply these poses yourself when you come to use your own meshes.

```
PxVehicleWheelsSimData* wheelsSimData = PxVehicleWheelsSimData::allocate(numWheels);

{

   //Compute the wheel center offsets from the origin.
   PxVec3 wheelCenterActorOffsets[PX_MAX_NB_WHEELS];
   const PxF32 frontZ = chassisDims.z*0.3f;
   const PxF32 rearZ = -chassisDims.z*0.3f;
   fourwheel::computeWheelCenterActorOffsets4W(frontZ, rearZ, chassisDims,
       wheelWidth, wheelRadius, numWheels, wheelCenterActorOffsets);

   //Set up the simulation data for all wheels.
   fourwheel::setupWheelsSimulationData(vehicle4WDesc.wheelMass,
       vehicle4WDesc.wheelMOI, wheelRadius, wheelWidth,
       numWheels, wheelCenterActorOffsets,
       vehicle4WDesc.chassisCMOffset, vehicle4WDesc.chassisMass,
       wheelsSimData);

}
```

Vehicle simulation setup

```
PxVehicleDriveSimData4W driveSimData;
//Differential
PxVehicleDifferential4WData diff;
diff.mType=PxVehicleDifferential4WData::eDIFF_TYPE_LS_4WD;
driveSimData.setDiffData(diff);
//Engine
PxVehicleEngineData engine;
engine.mPeakTorque=500.0f;
engine.mMaxOmega=600.0f;//approx 6000 rpm
driveSimData.setEngineData(engine);
//Gears
PxVehicleGearsData gears;
gears.mSwitchTime=0.5f;
driveSimData.setGearsData(gears);
//Clutch
PxVehicleClutchData clutch;
clutch.mStrength=10.0f;
driveSimData.setClutchData(clutch);
//Ackermann steer accuracy
PxVehicleAckermannGeometryData ackermann;
ackermann.mAccuracy=1.0f;
ackermann.mAxleSeparation=
     wheelsSimData->getWheelCentreOffset(PxVehicleDrive4WWheelOrder::eFRONT_LEFT).z-
     wheelsSimData->getWheelCentreOffset(PxVehicleDrive4WWheelOrder::eREAR_LEFT).z;
ackermann.mFrontWidth=
     wheelsSimData->getWheelCentreOffset(PxVehicleDrive4WWheelOrder::eFRONT_RIGHT).x

     wheelsSimData->getWheelCentreOffset(PxVehicleDrive4WWheelOrder::eFRONT_LEFT).x;
ackermann.mRearWidth=
     wheelsSimData->getWheelCentreOffset(PxVehicleDrive4WWheelOrder::eREAR_RIGHT).x

     wheelsSimData->getWheelCentreOffset(PxVehicleDrive4WWheelOrder::eREAR_LEFT).x;
```

Trust the sample code, the sample code is your friend. Some of these numbers could be quite tweakable.

Putting it all together

- This is the code which actually uses **createVehicle4W** to create a **PxVehicleDrive4W** object and add it to the scene.
- Note that the vehicle object owns a rigid dynamic actor, and that's what's added to the scene. The rest of the vehicle object is the data and state to do with the car simulation (engine, wheels, gearbox, etc).

```
VehicleDesc vehicleDesc = initVehicleDesc();

gVehicle4W = createVehicle4W(vehicleDesc, gPhysics, gCooking);

PxTransform startTransform(PxVec3(0, (vehicleDesc.chassisDims.y*0.5f
+
     vehicleDesc.wheelRadius + 1.0f), 0), PxQuat(PxIdentity));
gVehicle4W->getRigidDynamicActor()->setGlobalPose(startTransform);

gScene->addActor(*gVehicle4W->getRigidDynamicActor());
```

Per-frame actions

- The sample code puts the car through a series of manoevres – forward, backwards, turns, handbrake turns.
- It does this by populating **gVehicleInputData** with simulated keypresses or simulated analog stick data (based on whether **gMimicKeyInputs** is true or false). You should replace this code with actual inputs from the keyboard or gamepad.
- Then it applies this input data to the vehicle.
- Then it performs raycasts for each vehicle in the scene, to find out which wheels are on the ground.
- Then it updates each vehicle, using the raycast results. This function turns the wheels, updates the engine RPM, works out the car's current speed, and works out what forces to apply to the car model.
- Then it ticks the scene for $1/60^{th}$ of a second.
- This all happens in the function **stepPhysics()**

Rendering the scene

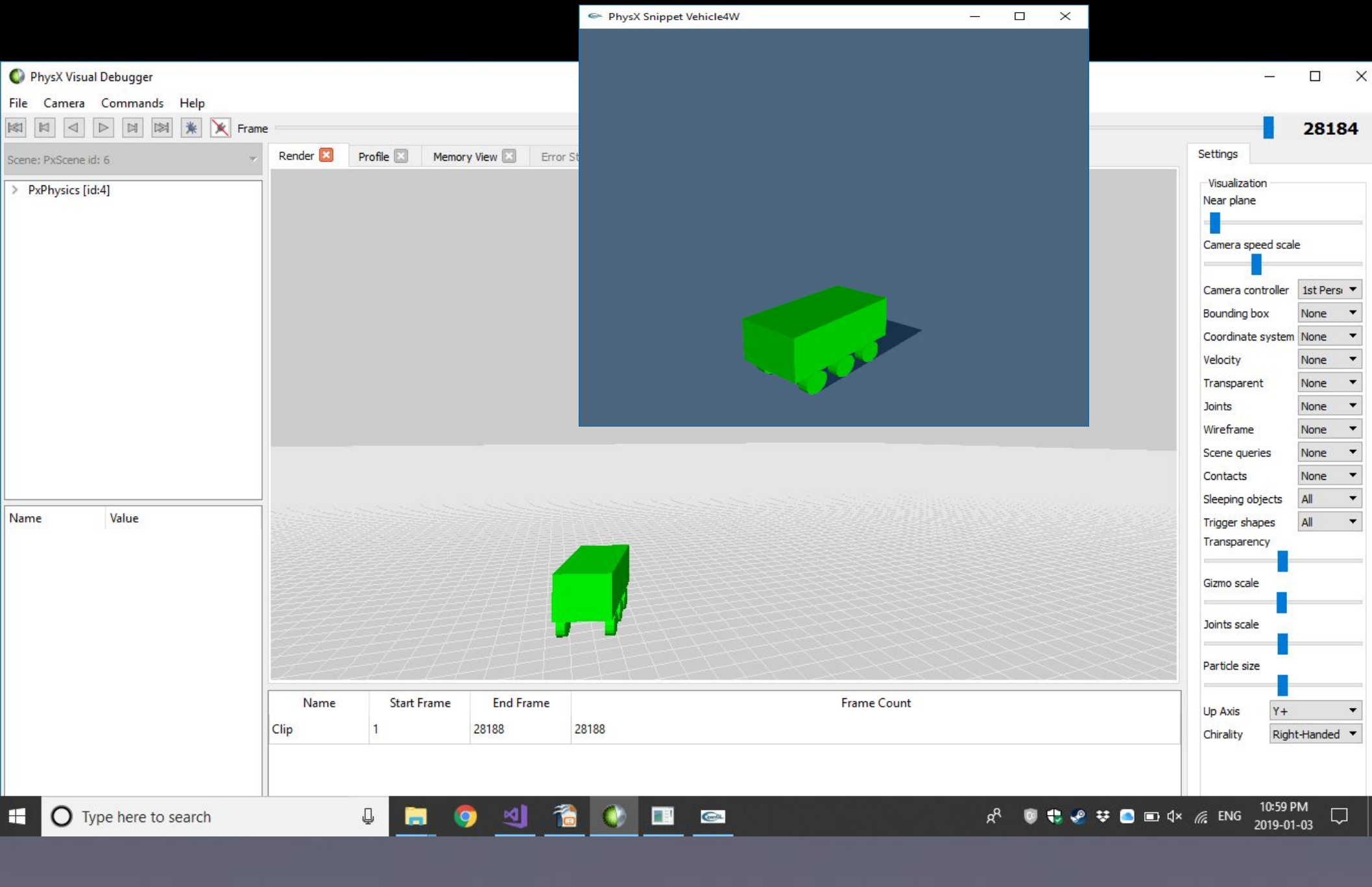
- The snippet samples will render the scene for you if **RENDER_SNIPPET** is defined
- Otherwise you can see the scene in PvD
- Here's the standard render code in **renderCallback()**

```
PxU32 nbActors = gScene->getNbActors(PxActorTypeFlag::eRIGID_DYNAMIC |
    PxActorTypeFlag::eRIGID_STATIC);
if(nbActors)
{
    std::vector<PxRigidActor*> actors(nbActors);
    gScene->getActors(PxActorTypeFlag::eRIGID_DYNAMIC | PxActorTypeFlag::eRIGID_STATIC,
        reinterpret_cast<PxActor**>(&actors[0]), nbActors);
    Snippets::renderActors(&actors[0], static_cast<PxU32>(actors.size()), true);
}
```

Debugging

- PvD – PhysX debugger
- Separate EXE
- Needs additional setup when you initialize PhysX (as shown earlier) so the scene will talk to the debugger
- Debugger window lists actors, renders shapes and meshes
- Can show additional data like forces, velocities, normals
- If your rendered scene doesn't basically match this, your rendering is wrong ;)
- Disabled in 'release' config of PhysX

Rendered scene and PvD view of scene



Debugging

- Implement own basic debugging with **getRenderBuffer()**
- See SnippetRender.cpp: **renderActors(...)**

```
// Still need to enable the visualization somehwere in init
gScene->setVisualizationParameter(PxVisualizationParameter::eSCALE,
1.0f);
gScene->setVisualizationParameter(PxVisualizationParameter::eACTOR_AX
ES, 1.0f);


// Somewhere during your render pass:
const PxRenderBuffer& rb = gScene->getRenderBuffer();
for(PxU32 i=0; i < rb.getNbLines(); i++)
{
    const PxDebugLine& line = rb.getLines()[i];
    // render the line in your application
    // drawLine(line.pos0.x, line.pos0.y, line.pos0.z, line.pos1.x,
line.pos1.y, line.pos0.z);
}
```

Integrating PhysX into your project

- Building the PhysX solution builds the libraries you'll need.
- Copy the libraries and header files into your game's repo and include / link them from there
- You'll need headers from physx/include and pxshared/include
- Also copy the DLLs. These have to be on the execution path when your game EXE runs (this usually means in the same folder)
- You'll find the built libraries and DLLs under physx\bin
- Release config is the fastest but doesn't support PvD. Profile supports PvD but doesn't check your data. Checked supports PvD and checks your data. Each config is correspondingly slower.

Things you can change

- Anywhere you see a hardcoded number in the sample code, it might be a worthwhile tweakable value
- Remember to adjust how many wheels your car has!
- Some values are documented, some are 'magic'
- Some numbers are not expressed in the units you might expect...

```
engine.mMaxOmega=600.0f;//approx 6000 rpm
   // This is because 'omega' is the symbol for rotational
   // velocity which PhysX measures in radians per second.
```

- The car and wheel meshes need to be replaced with your own meshes (sample ones are OK for milestone 2)
- Try other vehicle types – tanks, 6W drive (see SnippetVehicleCreate.cpp on creating other non-4W vehcile types vs SnippetVehicle4WCreate.cpp)

A word on collision...

- When the sample creates the car, it includes filter data which will be used by the scene's filter shader, indicating what kinds of surfaces the chassis and the wheels can collide with.
- Similarly when it creates the ground plane, this is flagged with the kinds of collision it will handle.
- If the wheels can collide with the ground, the car won't fall through it ;)
- You'll need to set the right flags for other objects that the car might 'collide' with (e.g. trigger volumes, pickups).
- For general collision information, read the PhysX help file. Basically you'll want to implement a collision callback, which will be called to inform you of each collision (and the two objects which collided).

Conclusions

- Use the PhysX vehicle SDK. It will save time and sanity.
- Mechanics of setup are complex - don’t be afraid to steal the PhysX sample code.
- The driving model in the PhysX sample app is now the baseline - need to make it better if you want good marks on the driving portion of things
- PhysX is generally very well documented. But also very extensive! Concentrate on what you need – mesh cooking, rigid body collisions, vehicles, raycasts.
- There are separate libraries for particles (including fluids) and for cloth if you're feeling incredibly ambitious, but don't get distracted ;)
- Remember to make a nice system for quickly reloading your tune-able variables and recreating your cars in PhysX – it'll save you lots of iteration time!