Chrono::Vehicle Tutorial

Chrono::Vehicle

- Chrono::Vehicle is a C++ middleware library for the modeling, simulation, and visualization of wheeled and tracked ground vehicles

- Chrono::Vehicle is a Chrono module, consisting of two libraries:
  - ChronoEngine_vehicle
    - Defines the system and subsystem base classes
    - Provides concrete, derived classes for instantiating templates from JSON specification files
    - Provides miscellaneous utility classes and free functions for file I/O, Irrlicht vehicle visualization, steering and speed controllers, vehicle and subsystem test rigs, etc.
  - ChronoModels_vehicle
    - Provides concrete classes for instantiating templates to model specific vehicle models
- Dependencies:
  - Chrono::Engine main module (required)
  - Chrono::Irrlicht and the Irrlicht library; Chrono::OpenGL and dependencies (optional)
  - Chrono::FEA and Chrono::MKL (optional)

Code design – systems and subsystems

- Systems are the functional blocks that participate in a co-simulation framework:
  - are isolated and separated
  - respect a well-defined communication data flow
  - can advance their state (dynamics) independently and asynchronously
  - Examples: powertrain, tire, terrain, driver, vehicle
- (Vehicle) Subsystems are functional elements in a vehicle model
  - have a particular functional role (the subsystem 'type')
  - a subsystem type can have many different implementations
  - Examples:
    - suspension, steering, driveline, brake, wheel
    - sprocket, idler, road-wheel, suspension, track shoe

Code design - templates

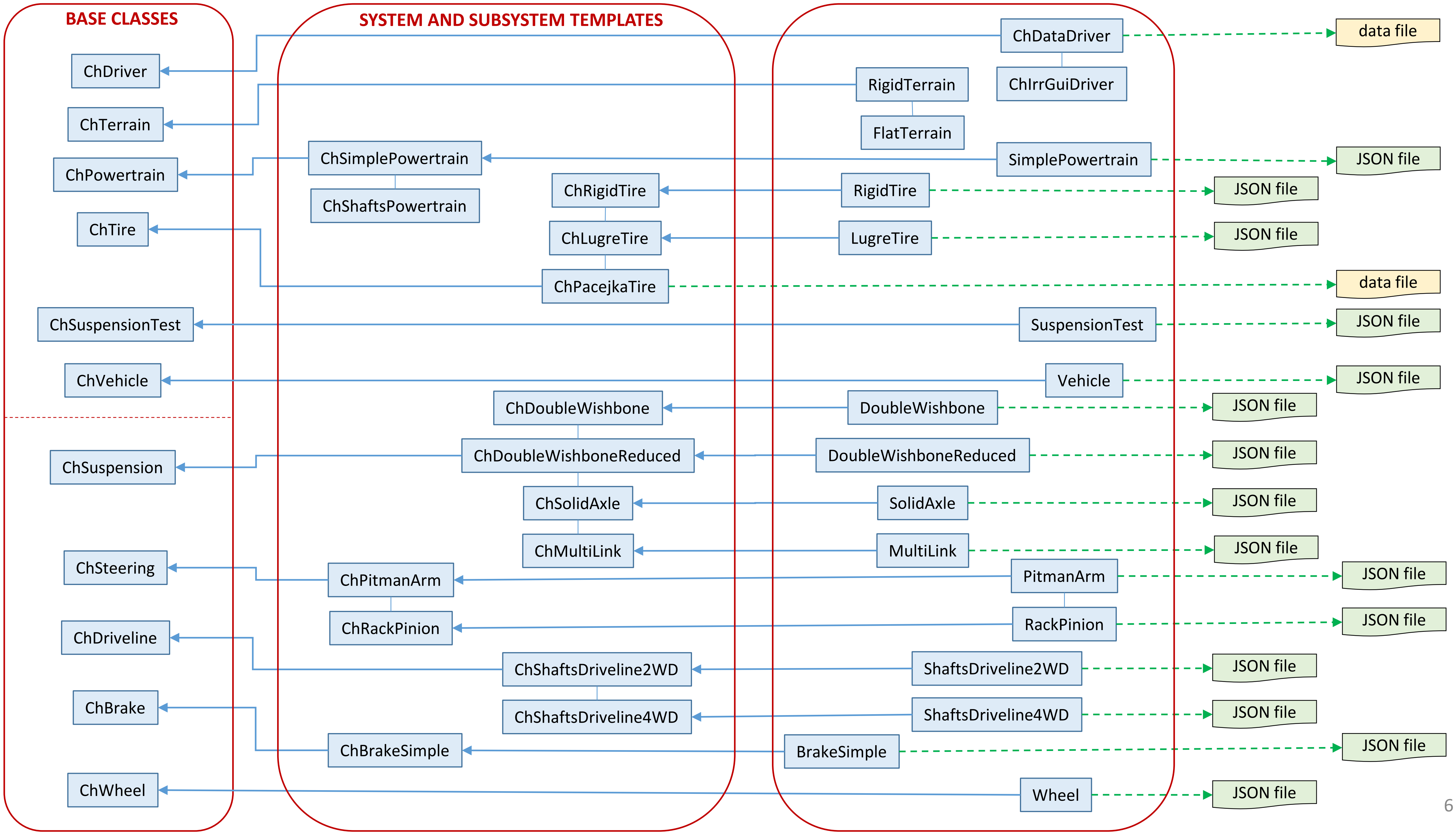
- Template-based modeling
  (not in the C++ sense)

- In Chrono::Vehicle, templates are parameterized models that define a particular implementation of a subsystem type:
  - Define the basic Chrono modeling elements (bodies, joints, force elements, etc.)
  - Impose the subsystem topology (connectivity)
  - Define the template parameters
  - Implement common functionality for the type of subsystem (e.g. ‘suspension’), particularized to the specific template (e.g. ‘double-wishbone’)

Code design – class hierarchy

- Chrono::Vehicle encapsulates templates for systems and subsystems in polymorphic C++ classes:
  - A base abstract class for the system/subsystem type (e.g. ChSuspension)
  - A derived, still abstract class for the system/subsystem template (e.g. ChDoubleWishbone)
  - Concrete class that particularize a given system/subsystem template (e.g. HMMWV_DoubleWishboneFront)

- Concrete classes:
  - User-defined – a derived class that satisfies all virtual functions imposed by the inherited template class
    - not part of the Chrono::Vehicle library
    - several example concrete classes (in the models library) and demo programs are provided
  - Generic – a derived class that satisfies all required virtual functions using parameter data from a specification file
    - part of the Chrono::Vehicle library
    - specification files use the JSON format



BASE CLASSES
SYSTEM AND SUBSYSTEM TEMPLATES
ChDriver
ChTerrain
ChPowertrain
ChTire
ChSuspensionTest
ChVehicle
ChSuspension
ChSteering
ChDriveline
ChBrake
ChWheel
ChSimplePowertrain
ChShaftsPowertrain
ChRigidTire
ChLugreTire
ChPacejkaTire
ChDoubleWishbone
ChDoubleWishboneReduced
ChSolidAxle
ChMultiLink
ChPitmanArm
ChRackPinion
ChShaftsDriveline2WD
ChShaftsDriveline4WD
ChBrakeSimple
ChDataDriver
ChIrrGuiDriver
RigidTerrain
FlatTerrain
SimplePowertrain
RigidTire
LugreTire
SuspensionTest
Vehicle
DoubleWishbone
DoubleWishboneReduced
SolidAxle
MultiLink
PitmanArm
RackPinion
ShaftsDriveline2WD
ShaftsDriveline4WD
BrakeSimple
Wheel
data file
JSON file

Code organization

FOLDER	CONTENT
data/vehicle	JSON specification files, visualization meshes, contact meshes
src/chrono_vehicle	Base system and subsystem class definitions (main Chrono::Vehicle library implementation)
src/chrono_models/vehicle	Concrete system and subsystem class definitions for specific vehicles
src/chrono_thirdparty/rapidjson	Clone of rapidjson – a JSON parser and generator library https://github.com/miloyip/rapidjson
src/demos/vehicle	Various demo programs (main drivers)

Note: additional, more complex, Chrono::Vehicle programs are available in the GitHub repository
https://github.com/projectchrono/chrono-projects

Code organization – vehicle subsystems

FOLDER	CONTENT
	Abstract base class definitions for systems and subsystems (ChVehicle, ChChassis, ChPowertrain, etc.)
chassis	Class definitions for chassis subsystem templates; concrete JSON-based implementations
driver	Driver system class definitions (ChDataDriver – file-based driver inputs; ChIrrGuiDriver – interactive driver inputs)
powertrain	Class definitions for powertrain subsystem templates; concrete JSON-based implementations
terrain	Terrain system class definitions (RigidTerrain, FlatTerrain)
tracked_vehicle	Subsystems for tracked vehicles
wheeled_vehicle	Subsystems for wheeled vehicles
utils	Various utility classes (controllers, vehicle visualization wrappers, etc.)

Code organization – wheeled vehicle subsystems

FOLDER	CONTENT
	Abstract base class definitions for systems and subsystems (ChWheeledVehicle, ChSuspension, ChTire, etc.)
antirollbar	Class definitions for antiroll bar templates; concrete JSON-based implementations
brake	Class definitions for brake subsystem templates; concrete JSON-based implementations
driveline	Class definitions for driveline subsystem templates; concrete JSON-based implementations
steering	Class definitions for steering subsystem templates; concrete JSON-based implementations
suspension	Class definitions for suspension subsystem templates; concrete JSON-based implementations
tire	Class definitions for tire system templates; concrete JSON-based implementations
vehicle	Concrete implementation of a JSON-based wheeled vehicle system template
wheel	Concrete implementation of a JSON-based wheel subsystem template
utils	Various utility classes (wheeled vehicle visualization wrappers, interactive driver, suspension test rig)

Code organization – tracked vehicle subsystems

FOLDER	CONTENT
	Abstract base class definitions for systems and subsystems (ChTrackedVehicle, ChSprocket, ChIdler, etc.)
brake	Class definitions for brake subsystem templates; concrete JSON-based implementations
driveline	Class definitions for driveline subsystem templates; concrete JSON-based implementations
idler	Class definitions for idler subsystem templates (with tensioner); concrete JSON-based implementations
road_wheel	Class definitions for road-wheel subsystem templates; concrete JSON-based implementations
sprocket	Class definitions for sprocket system templates; concrete JSON-based implementations
suspension	Class definitions for suspension assembly system templates; concrete JSON-based implementations
track_assembly	Class definitions for track assembly system templates; concrete JSON-based implementations
track_shoe	Class definitions for track-shoe system templates; concrete JSON-based implementations
vehicle	Concrete implementation of a JSON-based tracked vehicle subsystem template
utils	Various utility classes (tracked vehicle visualization wrappers, interactive driver, track test rig)

Code organization – models

FOLDER	CONTENT	
src/chrono_vehicle/	ChSubsysDefs.h	miscellaneous enums for model definition and creation
src/chrono_models/generic	Concrete implementations of system and subsystem classes for a generic vehicle	
src/chrono_models/hmmwv	Concrete implementations of system and subsystem classes for a HMMWV	
src/chrono_models/m113	Driver program for simulating a vehicle completely defined through JSON specification files	
data/vehicle	renderZ.pov	a generic POV-Ray script for frame rendering

Code organization – demos

FOLDER	CONTENT
demo_VEH_Articulated	Articulated wheeled vehicle (with trailer)
demo_VEH_DeformableSoil	Rigid wheel on SCM soil
demo_VEH_HMMWV	HMMWV vehicle (full double-wishbone suspension)
demo_VEH_HMMWV_DefSoil	HMMWV vehicle on SCM soil
demo_VEH_HMMWV9	HMMWV vehicle (reduced double-wishbone suspension)
demo_VEH_SteeringControler	Demonstration of PID steering and speed controllers (double-lane change)
demo_VEH_SuspensionTestRig	Suspension test rig defined through a JSON specification file
demo_VEH_WheeledGeneric	Generic wheeled vehicle (test bed for various templates)
demo_VEH_WheeledJSON	Vehicle completely defined through JSON specification files

FOLDER	CONTENT
demo_VEH_M113	M113 tracked vehicle on rigid terrain
demo_VEH_M113_DefSoil	M113 tracked vehicle on SCM soil
demo_VEH_M113_Parallel	M113 tracked vehicle with Chrono::Parallel

Code organization – test programs

Available in the GitHub repository https://github.com/projectchrono/chrono-projects

FOLDER	CONTENT
test_VEH_HMMWV_ANCFTire	HMMWV vehicle with deformable ANCF tires
test_VEH_HMMWV_Cosimulation	MPI co-simulation framework for vehicle with deformable tires on granular terrain
test_VEH_tirePacejka	test program for Pacjeka tire implementation
test_VEH_tireRig	tire test rig
test_VEH_tireRig_Cosimulation	MPI co-simulation framework for single deformable tire on granular terrain

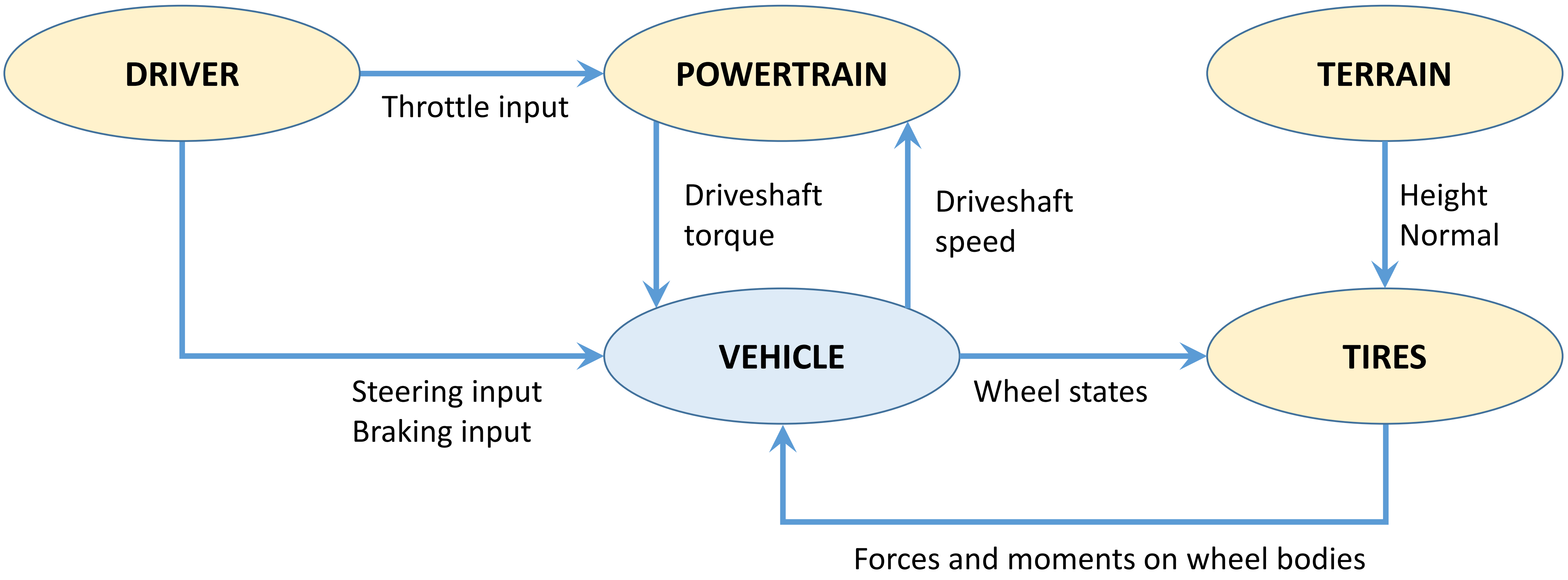
FOLDER	CONTENT
test_VEH_sprocketProfile	test for custom sprocket-track shoe contact processing

Simulation

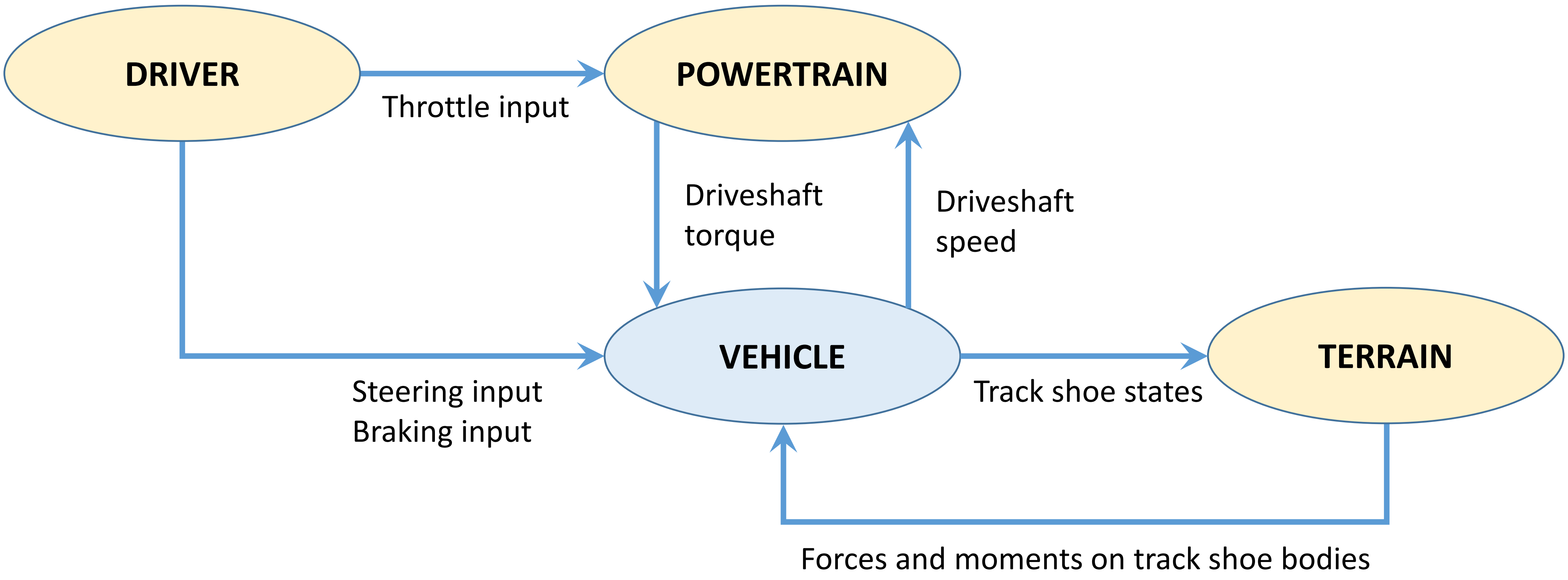
Inter-system communication

Skeleton of the simulation loop

Data flow (wheeled vehicles)

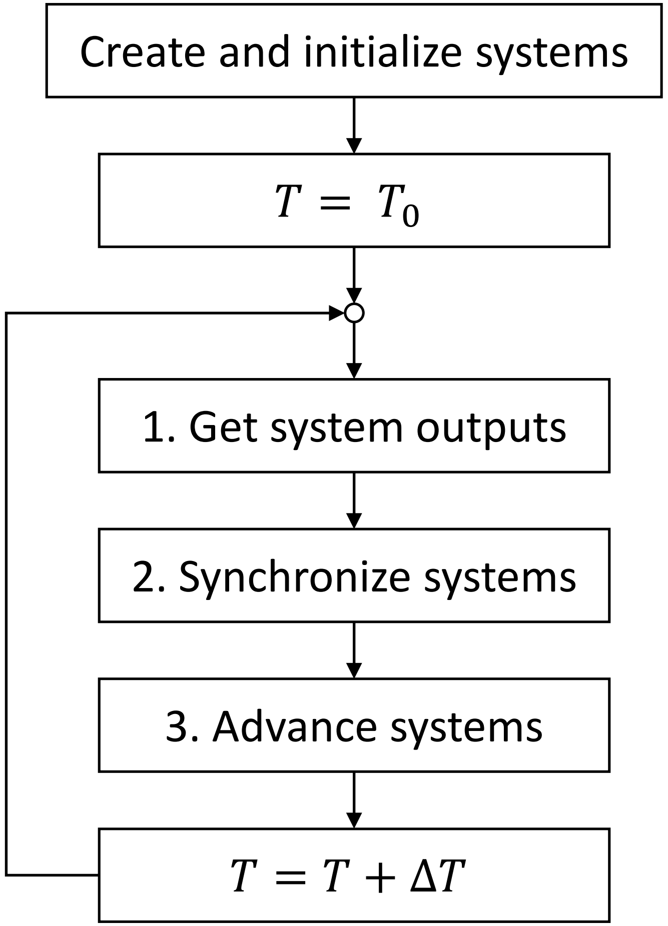


Data flow (tracked vehicles)



Simulation loop

- Framework: co-simulation with explicit coupling
- Systems advance between communication points asynchronously and at independent rates
- Between two successive communication points, each system extrapolates data from other systems



1. Get system outputs

- Outputs from each system are obtained from accessor methods (virtual functions declared by the corresponding base class)

```
// Collect output data from modules (for inter-module communication)
throttle_input = driver.GetThrottle();
steering_input = driver.GetSteering();
braking_input = driver.GetBraking();
powertrain_torque = powertrain.GetOutputTorque();
driveshaft_speed = vehicle.GetDriveshaftSpeed();
for (int i = 0; i < num_wheels; i++) {
  tire_forces[i] = tires[i]->GetTireForce();
  wheel_states[i] = vehicle.GetWheelState(i);
}
```

2. Synchronize systems

- Each system base class declares a virtual function Update() with a signature appropriate for the particular type of system

```
// Update modules (process inputs from other modules)
time = vehicle.GetChTime();
driver.Synchronize(time);
powertrain.Synchronize(time, throttle_input, driveshaft_speed);
vehicle.Synchronize(time, steering_input, braking_input, powertrain_torque, tire_forces);
terrain.Synchronize(time);
for (int i = 0; i < num_wheels; i++)
  tires[i]->Synchronize(time, wheel_states[i], terrain);
```

3. Advance systems

- Each system base class declares a virtual function Advance() with a single parameter, the time interval between two communication points (ΔT)
- A particular system may take as many intermediate steps (constant or variable step-size) as needed to advance the state of the system by ΔT. If the system has no internal dynamics, this function can be a no-op.

```
// Advance simulation for one timestep for all modules
double step = realtime_timer.SuggestSimulationStep(step_size);
driver.Advance(step);
powertrain.Advance(step);
vehicle.Advance(step);
terrain.Advance(step);
for (int i = 0; i < num_wheels; i++)
  tires[i]->Advance(step);
```

{JSON}

JavaScript Object Notation

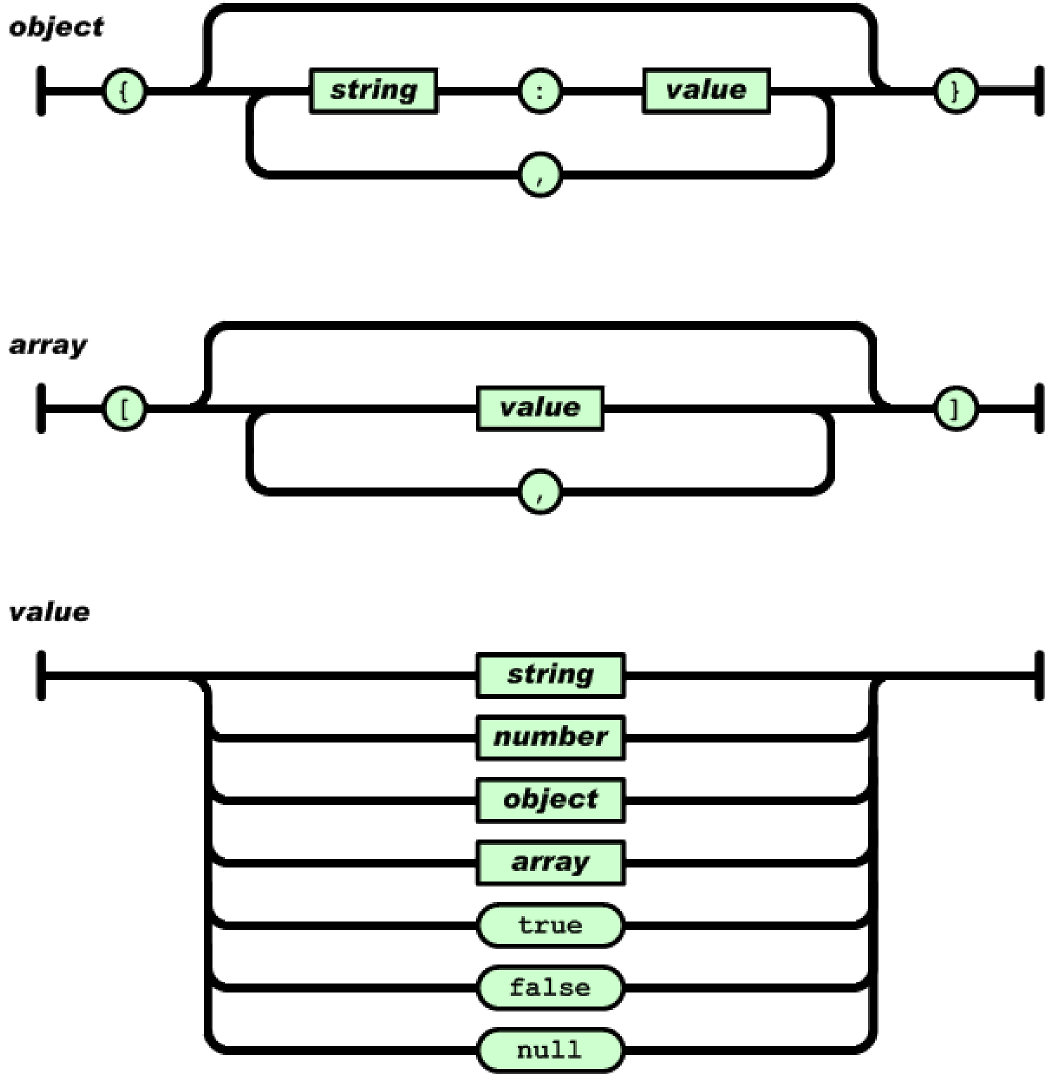
What is JSON?

- JavaScript Object Notation
- JSON is syntax for storing and exchanging text information. Much like XML.
- JSON is smaller than XML, faster and easier to parse.
- JSON:
  - JSON is a lightweight text-data interchange format
  - JSON is *language-independent* (the “JavaScript” in its name is misleading)
  - JSON is *"self-describing"* and easy to understand (that’s why it doesn’t even provide for comments!)
- Defined in [RFC 4627]
- [http://json.org/] has more information

Data types and syntax

- JSON’s basic types are:
  - Number: usually double precision floating-point
  - String: double-quoted
  - Boolean: true or false
  - Array: ordered sequence of values, comma-separated, enclosed in square brackets ‘[‘ and ‘]’
  - Object: unordered collection of key:value pairs, comma-separated, enclosed in curly braces ‘{‘ and ‘}’
  - null: empty
- Structural characters: [ ] { } : ,
- White spaces have no semantics around the structural characters

Very simple grammar



Example

key : value pair

string

number

key : value pair

object

array

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    },
    "phoneNumbers": [
        {
            "type": "home",
            "number": "212 555-1234“,
            “extension": 307
        },
        {
            "type": "fax",
            "number": "646 555-4567"
        }
    ]
}
```

Keys
Values
Structural characters

RapidJSON

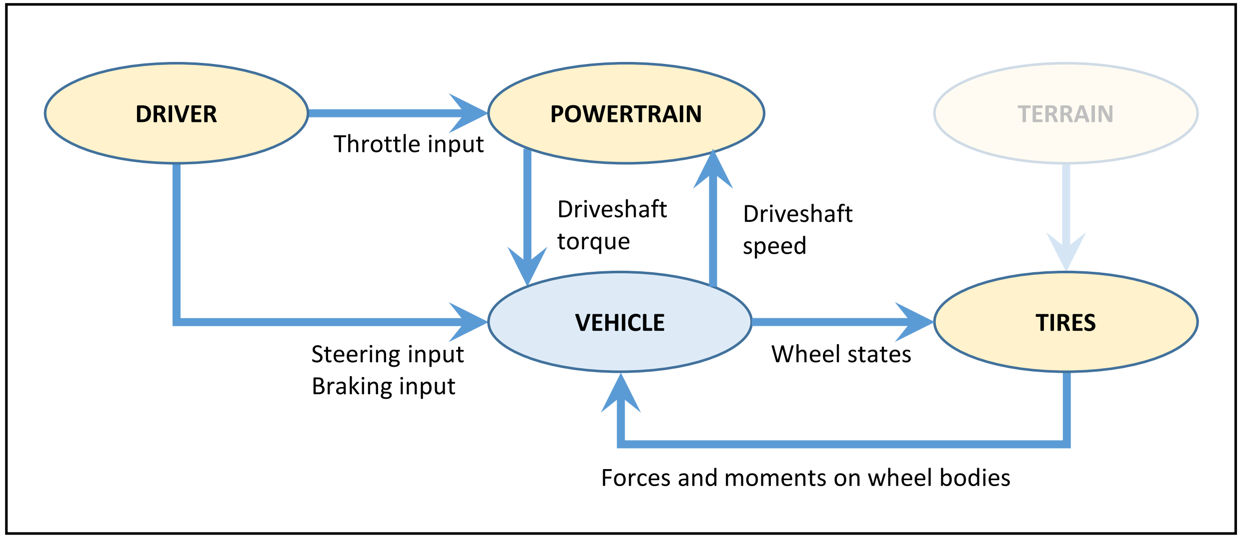
- Copyright (c) 2011-2014 Milo Yip (miloyip@gmail.com)
- RapidJSON is a JSON parser and generator for C++. It was inspired by RapidXml
- Available on GitHub: https://github.com/miloyip/rapidjson/
- Documentation: http://miloyip.github.io/rapidjson/
- RapidJSON is a header-only C++ library.

- The main RapidJSON headers are bundled in the chrono-T project

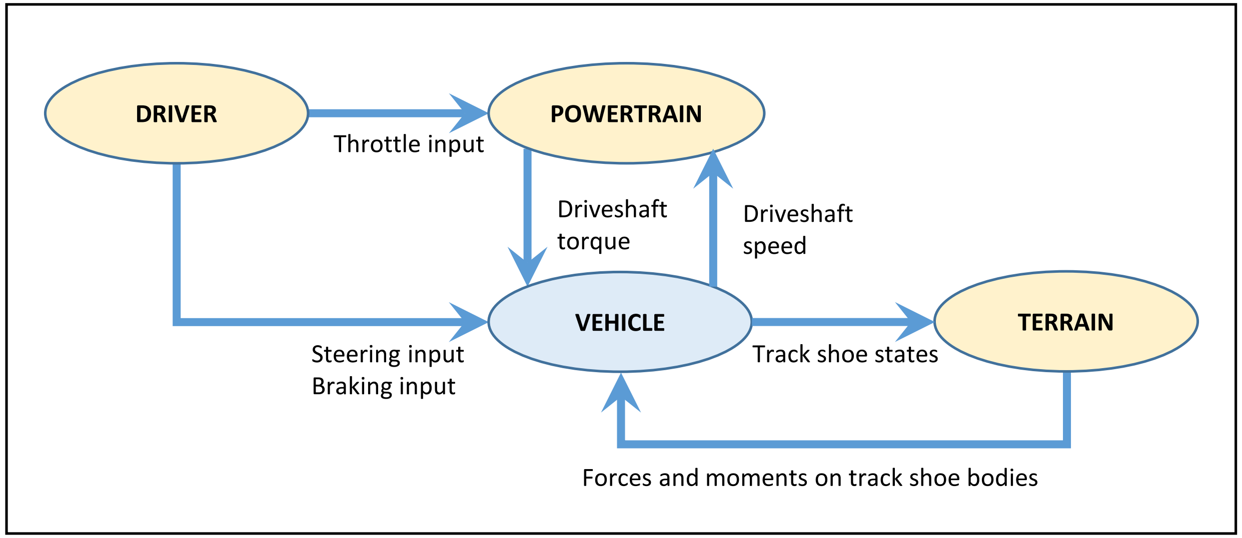
- NOTE: recently updated to latest RapidJSON version (1.1.0 – August 2016) which provides support for relaxed JSON syntax (support for single-line and multi-line C++-style comments)

Vehicle system
Chassis subsystem

Data flow

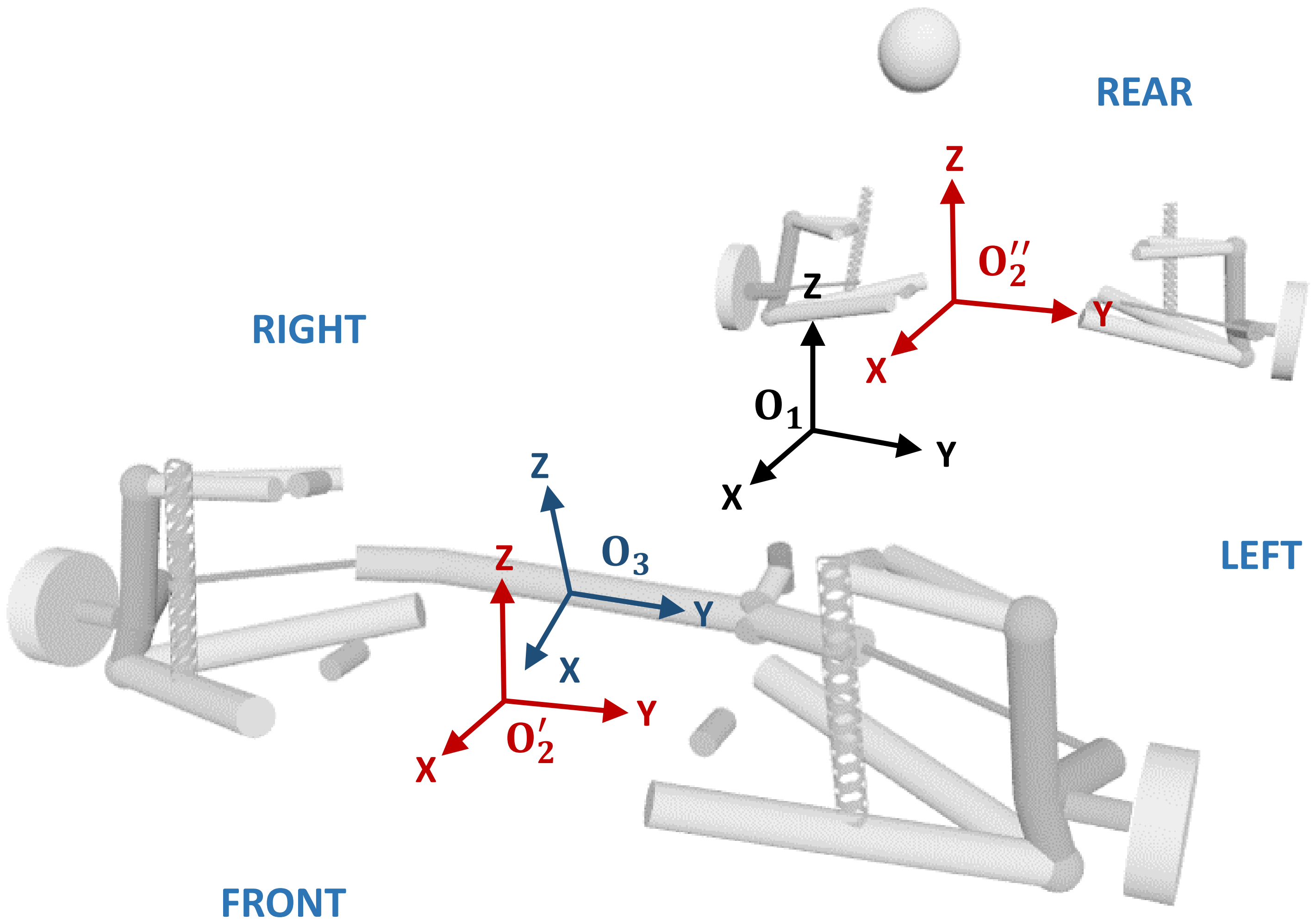


Wheeled vehicles



Tracked vehicles

Vehicle ISO reference frames



(XYZ) – chassis reference frame
(XYZ) – suspension reference frame
(XYZ) – steering reference frame

ChVehicle base class

```
/// Base class for chrono vehicle systems.
/// This class provides the interface between the vehicle system and other
/// systems (tires, driver, etc.)
class CH_VEHICLE_API ChVehicle
```

- A ChVehicle has:

```
ChSystem* m_system;                        ///< pointer to the Chrono system
std::shared_ptr<ChChassis> m_chassis;  ///< handle to the chassis subsystem
```

```
bool m_ownsSystem;    ///< true if system created at construction
double m_stepsize;    ///< integration step-size for the vehicle system
```

- NOTE: ChVehicle is an abstract base class with protected constructors.
- Only derived classes (ChWheeledVehicle and ChTrackedVehicle) can be instantiated

ChVehicle base class accessors

- Deferring to its constituent subsystems as needed, a ChVehicle provides accessors for:
  - Underlying ChSystem
  - Handle to the vehicle chassis
  - Chassis state (reference frame and COM)
  - Angular speed of the vehicle driveshaft (connection to powertrain)

- A ChVehicle intermediates communication between other systems (e.g., powertrain, driver, etc.) and constituent subsystems (e.g., suspensions, brakes, etc.)

ChVehicle base class virtual functions

```
/// Get a handle to the vehicle's driveshaft body.
virtual std::shared_ptr<ChShaft> GetDriveshaft() const = 0;

/// Get the angular speed of the driveshaft.
/// This function provides the interface between a vehicle system and a
/// powertrain system.
virtual double GetDriveshaftSpeed() const = 0;
```

- Initialize the vehicle at a specified position and orientation

```
/// Initialize this vehicle at the specified global location and orientation.
virtual void Initialize(const ChCoordsys<>& chassisPos  ///< [in] initial global position and orientation
                        double chassisFwdVel = 0        ///< [in] initial chassis forward velocity
                        ) = 0;
```

- Advance the state of the vehicle system to the next communication time

```
/// Advance the state of this vehicle by the specified time step.
virtual void Advance(double step);
```

ChChassis base class

- A ChChassis is a ChPart:

```
/// Base class for the chassis vehicle subsystem.
class CH_VEHICLE_API ChChassis : public ChPart
```

- A ChChassis has:

```
std::shared_ptr<ChBodyAuxRef> m_body;  ///< handle to the chassis body
bool m_fixed;                          ///< is the chassis body fixed to ground?
```

ChChassis base class accessors

- A ChChassis provides accessors for:
  - Chassis mass and inertia properties
  - Chassis state (reference frame and COM)
  - Vehicle speed (reference frame and COM)
  - Driver position (local and absolute)
  - Absolute acceleration of a point specified in local reference frame
- Any ChVehicle has a ChChassis

ChChassis base class virtual functions

- Specify mass and inertia properties of chassis body

```
/// Get the chassis mass.
virtual double GetMass() const = 0;

/// Get the moments of inertia of the chassis body.
virtual const ChVector<>& GetInertia() const = 0;

/// Get the location of the center of mass in the chassis frame.
virtual const ChVector<>& GetLocalPosCOM() const = 0;
```

- Specify local position and orientation of driver

```
/// Get the local driver position and orientation.
/// This is a coordinate system relative to the chassis reference frame.
virtual ChCoordsys<> GetLocalDriverCoordsys() const = 0;
```

- Initialize chassis within specified system, at given position and forward velocity

```
/// Initialize the chassis at the specified global position and orientation.
virtual void Initialize(ChSystem* system,                ///< [in] containing system
                        const ChCoordsys<>& chassisPos,  ///< [in] absolute chassis position
                        double chassisFwdVel             ///< [in] initial chassis forward velocity
                        );
```

JSON specification file for a chassis

```
{
  "Name":     "HMMWV chassis",
  "Type":     "Chassis",
  "Template": "RigidChassis",

  "Mass":     2086.52,
  "COM":      [0.056, 0, 0.523],
  "Inertia":  [1078.52, 2955.66, 3570.20],

   "Driver Position":
  {
    "Location":     [0, 0.5, 1.2],
    "Orientation":  [1, 0, 0, 0]
  },

  "Visualization":
  {
    "Mesh Filename":  "hmmwv/hmmwv_chassis.obj",
    "Mesh Name":      "hmmwv_chassis_POV_geom"
  }
}
```

System type (string)

Template type (string)

Offset of the chassis COM with respect to the chassis reference frame.

Chassis moments of inertia with respect to the chassis centroidal frame.

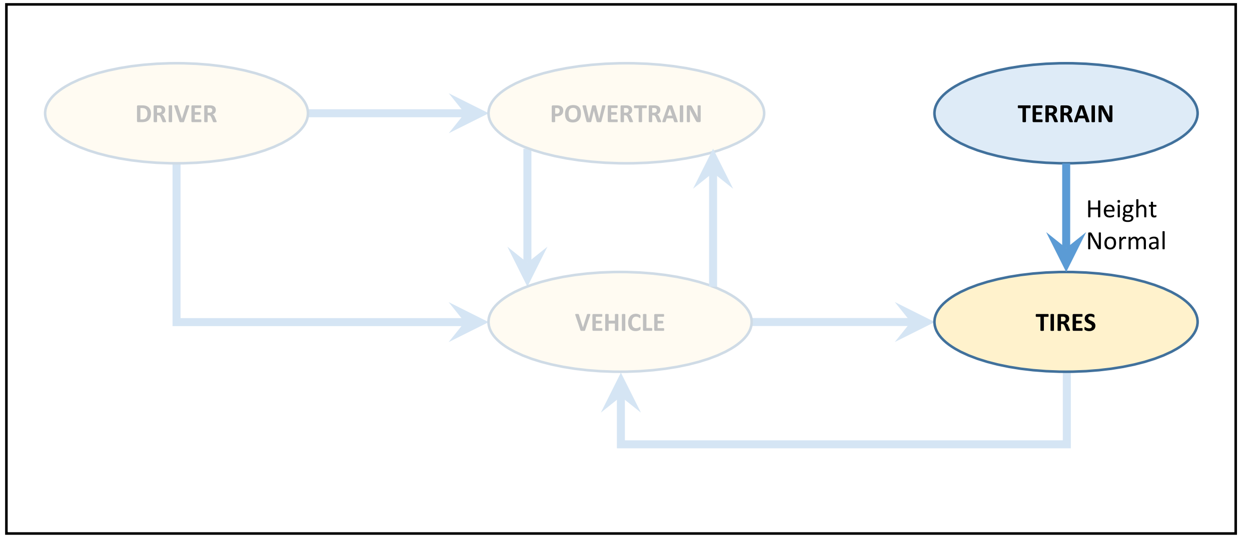
Position and orientation of driver with respect to the chassis reference frame.

Name of a Waveform OBJ file

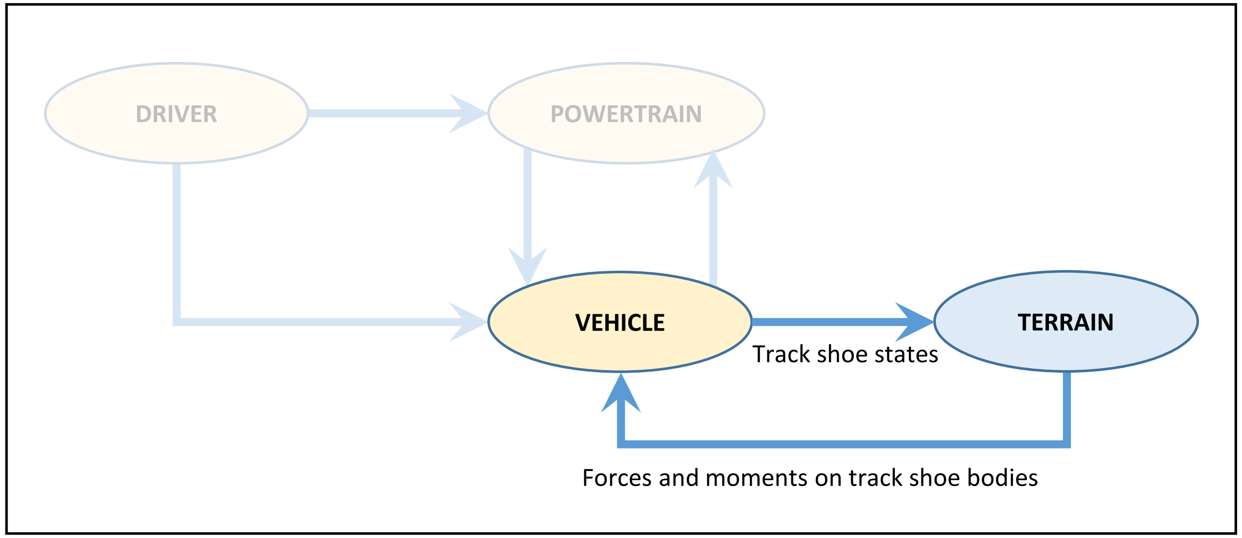
Name of a PovRay input file (mesh_name.inp) which defines a PovRay object (mesh_name)

Terrain Models

Data flow



Wheeled vehicles



Tracked vehicles

ChTerrain base class

- Defines the common interface for any terrain system
- All classes defining a particular terrain model inherit from ChTerrain

```
///
/// Base class for a terrain system.
///
class CH_VEHICLE_API ChTerrain
```

ChTerrain base class virtual methods

- Synchronize() and Advance() – common to all subsystems
  - Typically not used (overridden) for non-deformable terrains

- Return terrain height and normal direction

```
/// Get the terrain height at the specified (x,y) location.
virtual double GetHeight(double x, double y) const = 0;

/// Get the terrain normal at the specified (x,y) location.
virtual ChVector<> GetNormal(double x, double y) const = 0;
```

Rigid terrain: RigidTerrain class

- RigidTerrain is a concrete class
- The terrain is modeled as a rigid body (ground) with an attached contact shape modeled as:
  - A box or multiple side-by-side boxes (tiled)
  - A triangular mesh (provided as a Wavefront OBJ file)
  - A height-map (provided as a gray-scale BMP image)
- A RigidTerrain object can be constructed:
  - Programatically (see demo_VEH_HMMWV.cpp)
  - From a JSON specification file

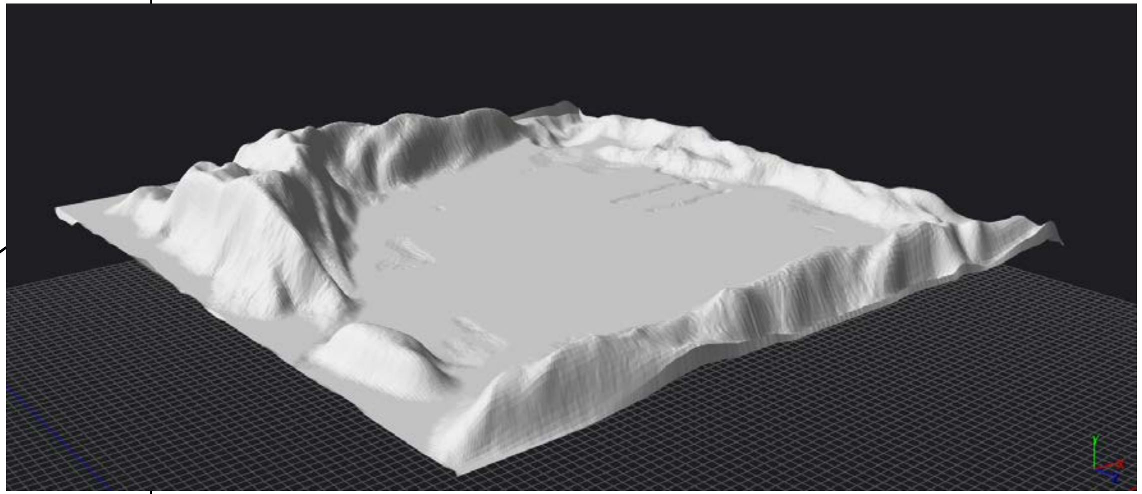
```
/// Rigid terrain model.
/// This class implements a terrain modeled as a rigid shape which can interact
/// through contact and friction with any other bodies whose contact flag is
/// enabled. In particular, this type of terrain can be used in conjunction with
/// a ChRigidTire.
class CH_VEHICLE_API RigidTerrain : public ChTerrain
```

RigidTerrain initialization functions

```
/// Initialize the terrain system (flat).
/// This version uses a rigid box of specified dimensions and with specified
/// material properties. If tiled = true, multiple side-by-side boxes are used.
void Initialize(double height,            ///< [in] terrain height
                double sizeX,             ///< [in] terrain dimension in the X direction
                double sizeY,             ///< [in] terrain dimension in the Y direction
                bool tiled = false,       ///< [in] terrain created from multiple tiled boxes
                double max_tile_size = 1  ///< [in] maximum tile size
                );

/// Initialize the terrain system (mesh).
/// this version uses the specified mesh, for both visualization and contact.
void Initialize(const std::string& mesh_file,   ///< [in] filename of the input mesh (OBJ)
                const std::string& mesh_name,   ///< [in] name of the mesh asset
                double sweep_sphere_radius = 0  ///< [in] radius of sweep sphere
                );

/// Initialize the terrain system (height map).
/// This version uses the specified BMP file as a height map to create a mesh for
/// both contact and visualization.
void Initialize(const std::string& heightmap_file,  ///< [in] filename for the height map (BMP)
                const std::string& mesh_name,       ///< [in] name of the mesh asset
                double sizeX,                       ///< [in] terrain dimension in the X direction
                double sizeY,                       ///< [in] terrain dimension in the Y direction
                double hMin,                        ///< [in] minimum height (black level)
                double hMax                         ///< [in] maximum height (white level)
                );
```

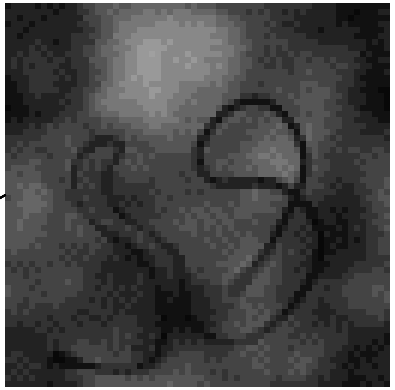
JSON specification file for RigidTerrain (mesh)

```
{
  "Name":     "Rigid plane",
  "Type":     "Terrain",
  "Template": "RigidTerrain",

  "Contact Material":
  {
      "Coefficient of Friction":    0.9,
      "Coefficient of Restitution": 0.01,

      "Properties": {
          "Young Modulus":          2e7,
          "Poisson Ratio":          0.3
      },

      "Coefficients": {
          "Normal Stiffness":       2e5,
          "Normal Damping":         40.0,
          "Tangential Stiffness":   2e5,
          "Tangential Damping":     20.0
      }
  },

  "Geometry":
  {
    "Mesh Filename":  "terrain/meshes/test.obj",
    "Mesh Name":      "terrain_test_POV_geom"
  },

  "Visualization":
  {
    "Color": [0.5, 0.5, 0.8],
    "Texture File": "terrain/textures/dirt.jpg",
    "Texture Scaling": [200, 200]
  }
}
```



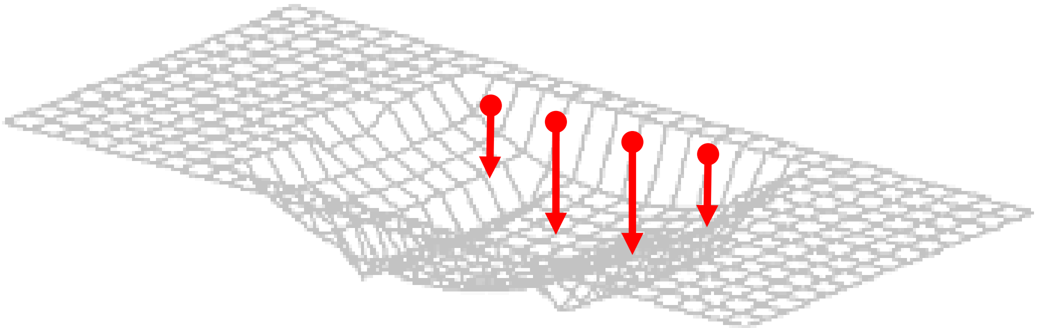
JSON specification file for RigidTerrain (height-map)

```json
{
  "Name":     "Rigid plane",
  "Type":     "Terrain",
  "Template": "RigidTerrain",

  "Contact Material":
  {
      "Coefficient of Friction":    0.9,
      "Coefficient of Restitution": 0.01,

      "Properties": {
          "Young Modulus":          2e7,
          "Poisson Ratio":          0.3
      },

      "Coefficients": {
          "Normal Stiffness":       2e5,
          "Normal Damping":         40.0,
          "Tangential Stiffness":   2e5,
          "Tangential Damping":     20.0
      }
  },

  "Geometry":
  {
    "Height Map Filename":  "terrain/height_maps/test64.bmp",
    "Mesh Name":            "terrain_test64_POV_geom",
    "Size":                 [128, 128],
    "Height Range":         [0, 4]
  },

  "Visualization":
  {
    "Color": [1.0, 1.0, 1.0],
    "Texture File": "terrain/textures/grass.jpg",
    "Texture Scaling": [16, 16]
  }
}
```



SCM terrain: DeformableTerrain class

- DeformableTerrain is a concrete class
- The terrain is modeled using a mesh
- The deformation of the mesh is along vertical direction only:
- The initial undeformed mesh can be created as:
  - A regular tiled mesh (filling a flat rectangle)
  - A triangular mesh (provided as a Wavefront OBJ file)
  - A height-map (provided as a gray-scale BMP image)
- A RigidTerrain object can be constructed programmatically (see demo_VEH_DeformableSoil.cpp)
- Based on the SCM Soil Contact Model [Krenn & Hirzinger (DLR), 2009]



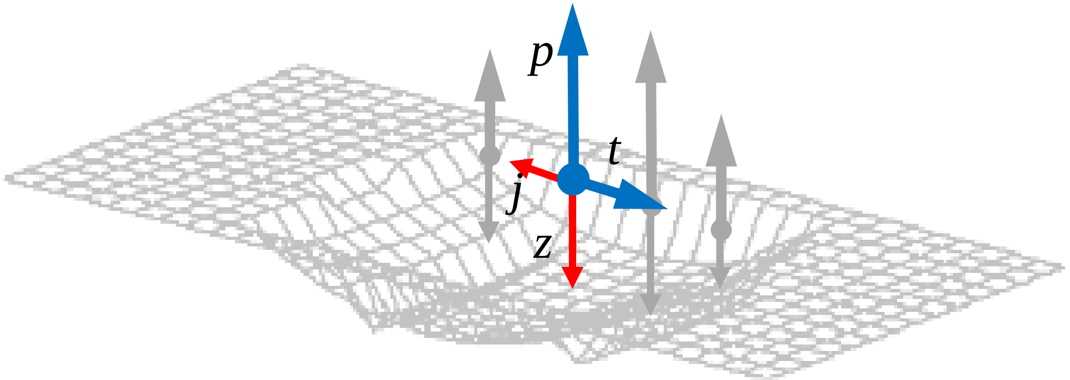
```
/// Deformable terrain model.
/// This class implements a terrain with variable heightmap. Unlike RigidTerrain, the vertical
/// coordinates of this terrain mesh can be deformed because of interaction with ground vehicles.
class CH_VEHICLE_API DeformableTerrain : public ChTerrain
```

DeformableTerrain: Chrono SCM Soil Contact Model

- The SCM model draws on the semi-empirical Bekker-Wong theory
- Pressure p related to sinkage z:

$$p = \left(\frac{k_c}{b} + k_\varphi \right) z^n$$

  - Parameters: K_φ … ker-Wong
  - K_c has negligible impact



- Tangential stress t given by Janosi-Hanamoto:

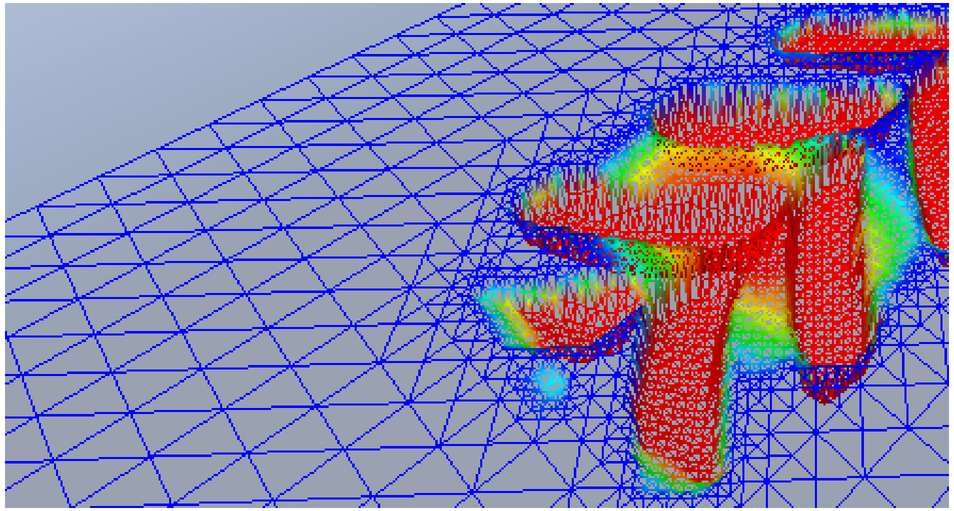
$$t = t_{max} \left(1 - e^{-j/k}\right)$$

$$t_{max} = c + p \tan(\varphi)$$

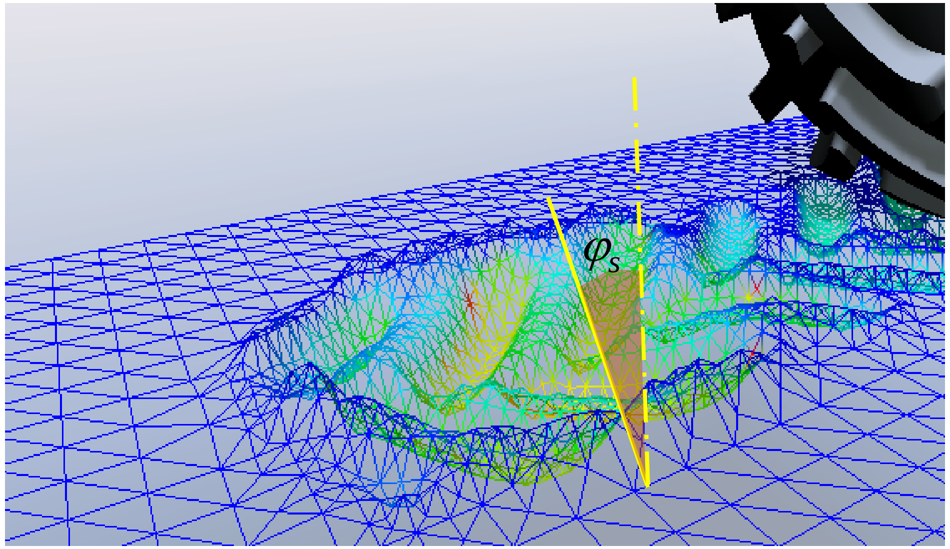
  - j is accumulated shear
  - Parameters: c cohesion, φ internal friction angle (Mohr theory), k Janosi parameter

DeformableTerrain: Chrono SCM Soil Contact Model customization

- The mesh can be automatically refined
  - Not limited to regular quadrilateral grids as in original SCM
  - Parameter: max. triangle size s



- Heuristic build-up of material at the boundary of the footprint
  - use a topological smoothing operator
  - Parameter: φ_s slope of rut and built-up material
  - φ_s can be $\mathrm{atan}(\mu_s)$ with μ_s is internal friction
  - Parameter: v percentual of material being displaced (100% means isochoric material)
  - Limitation: it cannot really simulate horizontal bulldozing effects like in a bulldozer blade



DeformableTerrain initialization functions

```
/// Initialize the terrain system (flat).
/// This version creates a flat array of points.
void Initialize(double height,  ///< [in] terrain height
                double sizeX,   ///< [in] terrain dimension in the X direction
                double sizeY,   ///< [in] terrain dimension in the Y direction
                int divX,       ///< [in] number of divisions in the X direction
                int divY        ///< [in] number of divisions in the Y direction
                );

/// Initialize the terrain system (mesh).
/// The initial undeformed mesh is provided via a Wavefront .obj file.
void Initialize(const std::string& mesh_file  ///< [in] filename of the input mesh (.OBJ file in Wavefront format)
                );

/// Initialize the terrain system (height map).
/// The initial undeformed mesh is provided via the specified BMP file as a height map
void Initialize(const std::string& heightmap_file,  ///< [in] filename for the height map (BMP)
                const std::string& mesh_name,       ///< [in] name of the mesh asset
                double sizeX,                       ///< [in] terrain dimension in the X direction
                double sizeY,                       ///< [in] terrain dimension in the Y direction
                double hMin,                        ///< [in] minimum height (black level)
                double hMax                         ///< [in] maximum height (white level)
                );
```

DeformableTerrain : example

```
vehicle::DeformableTerrain mterrain(&my_system);

// Optionally, displace/tilt/rotate the terrain reference plane:
mterrain.SetPlane(ChCoordsys<>(ChVector<>(0, 0, 0.5)));

// Initialize the geometry of the soil: use a regular grid:
mterrain.Initialize(0.2,1.5,5,20,60);

// Set the soil terramechanical parameters:
mterrain.SetSoilParametersSCM(1.2e6, // Bekker Kphi
                              0,     // Bekker Kc
                              1.1,   // Bekker n exponent
                              0,     // Mohr cohesive limit (Pa)
                              30,    // Mohr friction limit (degrees)
                              0.01,// Janosi shear coefficient (m)
                              5e7    // Elastic stiffness (Pa/m), before plastic yeld, must be > Kphi
                              );
mterrain.SetBulldozingFlow(true);    // inflate soil at the border of the rut
mterrain.SetBulldozingParameters(55, // slope of erosion at the border of the rut
                              0.8, // displaced material vs downward pressed material.
                              5,   // number of erosion refinements per timestep
                              10); // number of concentric vertex selections subject to erosion
// Turn on the automatic level of detail refinement
mterrain.SetAutomaticRefinement(true);
mterrain.SetAutomaticRefinementResolution(0.08);
```

FEA soil: FEADeformableTerrain class

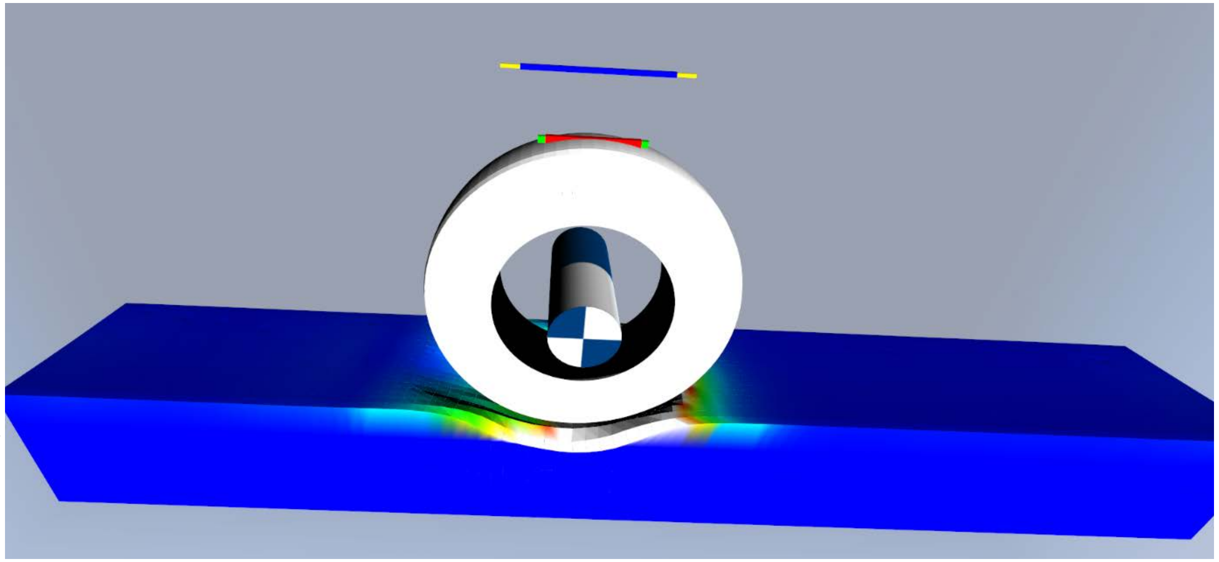
- FEADeformableTerrain is a concrete class
- Provides an easy way to construct a box of brick9 elements using a Drucker-Prager plasticity formulation for vehicle/soil interaction
- The terrain is modeled as a constrained box (sides and bottom)
  - Its contact shape may be modeled as a triangular mesh or a node cloud
  - Dimensions and material properties are passed to the initialize method
- An FEADeformableTerrain object can be constructed:
  - In the driver program, i.e. coding (see test_VEH_tireRig.cpp)
  - To be developed: JSON specification file

```
/// FEA Deformable terrain model.
/// This class implements a terrain made up of isoparametric finite elements. It features
/// Drucker-Prager plasticity and capped Drucker-Prager plasticity.
class CH_VEHICLE_API FEADeformableTerrain : public ChTerrain {
```

FEA Soil – Chrono implementation

Implementation of this class includes

- Discretization of a box (terrain) into a user-prescribed number of brick9 elements
    - Creation of corresponding nodes and elements
- Assignment of material properties to brick elements, including: density, modulus of elasticity, Poisson ratio, yield stress, hardening slope, dilatancy angle, and friction angle.
- Addition of assets for Irrlicht visualization



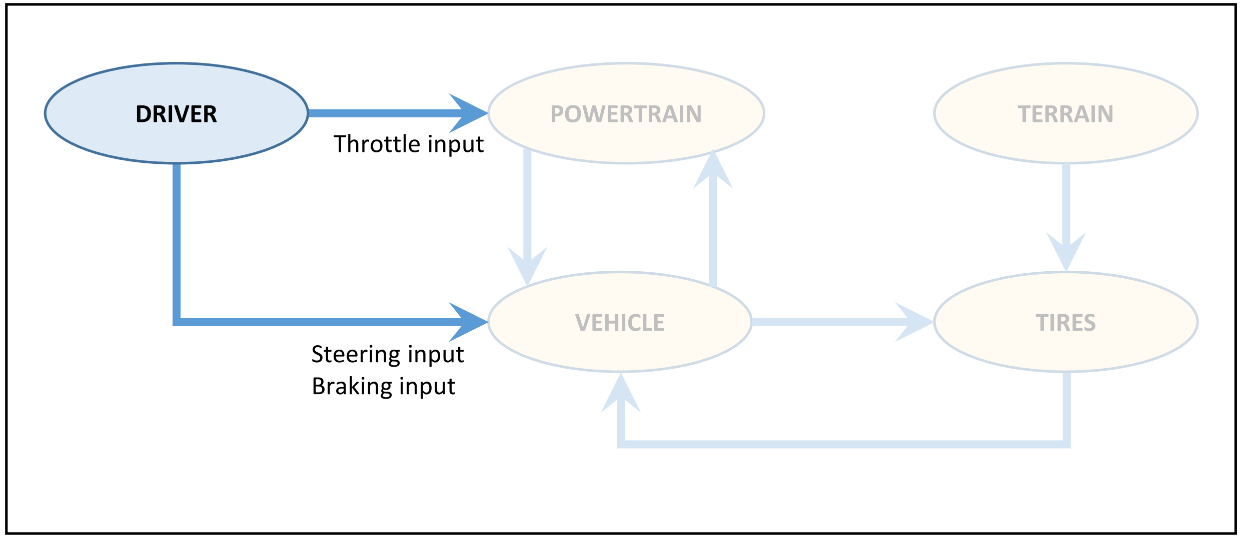
ANCF tire on a brick9, plastic FEA mesh of 100x20x4 elements

FEADeformableTerrain initialization functions

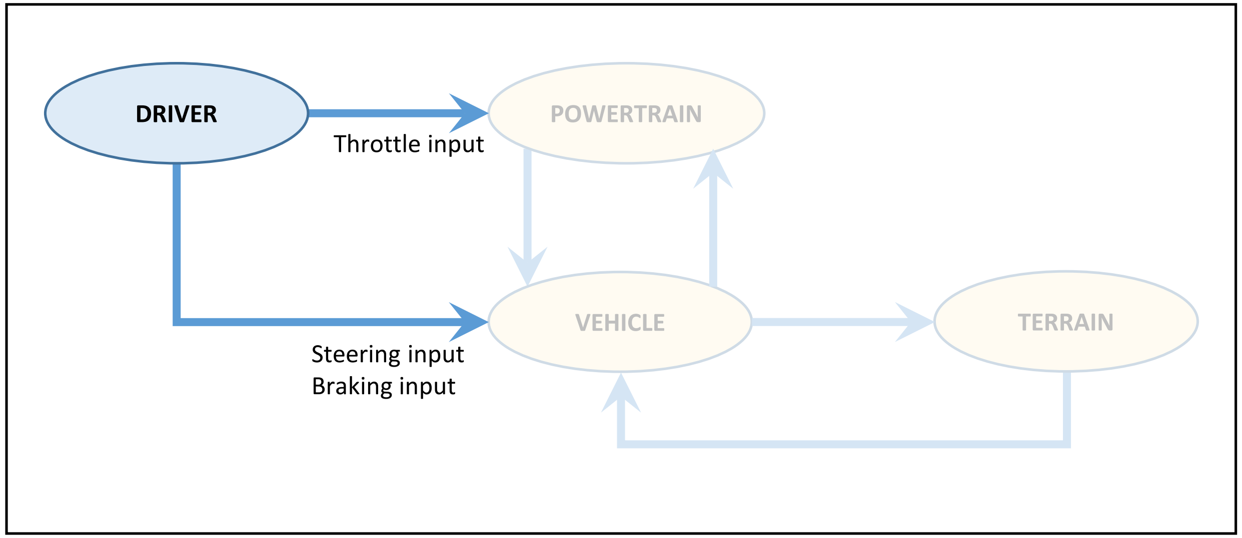
```
/// Set the properties of the Drucker-Prager FEA soil.
void SetSoilParametersFEA(double rho,              ///< [in] Soil density
                          double Emod,             ///< [in] Soil modulus of elasticity
                          double nu,               ///< [in] Soil Poisson ratio
                          double yield_stress,     ///< [in] Soil yield stress, for plasticity
                          double hardening_slope,  ///< [in] Soil hardening slope, for plasticity
                          double friction_angle,   ///< [in] Soil internal friction angle
                          double dilatancy_angle   ///< [in] Soil dilatancy angle
                          );

/// Initialize the terrain system (flat).
/// This version creates a flat array of points.
void Initialize(
    const ChVector<>& start_point,                 ///< [in] Base point to build terrain box
    const ChVector<>& terrain_dimension,           ///< [in] terrain dimensions in the 3 directions
    const ChVector<int>& terrain_discretization);  ///< [in] Number of finite elements in the 3 directions
```

Driver Models

Data flow



Wheeled vehicles



Tracked vehicles

ChDriver base class

- Defines the common interface for any driver system
  - A driver system can be open-loop or closed-loop (controller)
- All classes defining a particular driver model inherit from ChDriver

```
///
/// Base class for a vehicle driver system.
/// A driver system must be able to report the current values of the inputs
/// (throttle, steering, braking). A concrete driver class must set the member
/// variables m_throttle, m_steering, and m_braking.
///
class CH_VEHICLE_API ChDriver
```

ChDriver base class members and functions

- A ChDriver has:

```
ChVehicle& m_vehicle;  ///< reference to associated vehicle
double m_throttle;     ///< current value of throttle input
double m_steering;     ///< current value of steering input
double m_braking;      ///< current value of braking input
```

- ChDriver provides accessors for driver inputs (steering, throttle, and braking) and functions to record them in a file:

```
/// Initialize output file for recording driver inputs.
bool LogInit(const std::string& filename);

/// Record the current driver inputs to the log file.
bool Log(double time);
```

ChDriver base class virtual methods

- Synchronize the driver at a communication time with data from other systems.
  A concrete driver class may override the default no-op implementation (see ChDataDriver)

```
/// Update the state of this driver system at the current time.
virtual void Synchronize(double time) {}
```

- Advance the state of the tire to the next communication point.
  A concrete driver class may override the default no-op implementation (see ChIrrGuiDriver)

```
/// Advance the state of this driver system by the specified time step.
virtual void Advance(double step) {}
```

ChIrrGuiDriver

- Concrete driver system for interactive vehicle simulation
- Implemented as an Irrlicht event receiver, it reads the keyboard (A,W,S,D) to generate driver inputs
- Provides additional support for:
  - Vehicle tracking camera
  - Optional engine sound (using the IrrKlang library)
  - Rendering of joints, springs
  - Displaying vehicle stats
- Further specializations for wheeled and tracked vehicles are provided

```
/// Interactive driver model using keyboard inputs.
/// Irrlicht-based GUI driver for the a vehicle. This class implements the
/// functionality required by its base ChDriver class using keyboard inputs.
/// As an Irrlicht event receiver, its OnEvent() callback is used to keep track
/// and update the current driver inputs. As such it does not need to override
/// the default no-op Advance() virtual method.
class CH_VEHICLE_API ChIrrGuiDriver : public ChDriver, public irr::IEventReceiver
```

ChDataDriver

- Driver model based on inputs provided as time series:
  - Programmatically – a vector of 4-tuples {time, steering, throttle, braking}
  - From data file – ASCII file with a 4-tuple per line
- Time values must be unique
- If time values are not in ascending order, this must be indicated at construction
- Values at intermediate times are obtained through linear interpolation

```
/// Driver inputs from data file.
/// A driver model based on user inputs provided as time series. If provided as a
/// text file, each line in the file must contain 4 values:
///   time steering throttle braking
/// It is assumed that the time values are unique.
/// If the time values are not sorted, this must be specified at construction.
/// Driver inputs at intermediate times are obtained through linear interpolation.
class CH_VEHICLE_API ChDataDriver : public ChDriver
```

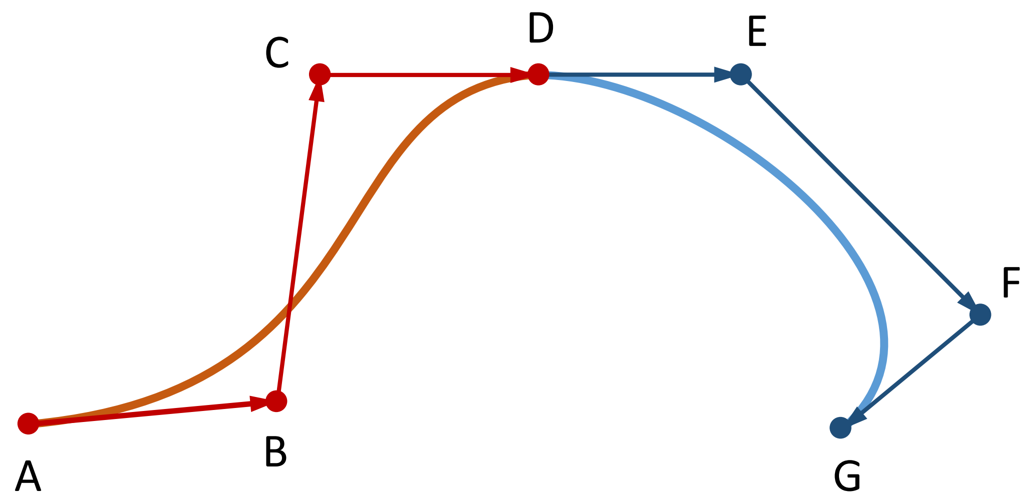
ChPathFollowerDriver

- Driver model based on inputs provided by two controllers:
  - Path steering controller – controls steering to follow prescribed path
  - Speed controller – controls throttle/brake to maintain constant speed
- Path specified as a piece-wise cubic Bezier curve
  - ChBezierCurve provides support for interpolation and visualization
- PID controllers specified through their gains

```
/// Closed-loop path-follower driver model.
/// A driver model that uses a path steering controller and a speed controller.
/// The steering controller adjusts the steering input to follow the prescribed
/// path.  The output from the speed controller is used to adjust throttle and
/// braking inputs in order to maintain the prescribed constant vehicle speed.
class CH_VEHICLE_API ChPathFollowerDriver : public ChDriver
```

ChBezierCurve

- Specified as a sequence of nodes and control points
- Match vectors for C^1 continuity



- Specification of points in input file (node, in, out):
  A A B
  D C E
  …
  G F G

ChPathFollowerDriver constructors

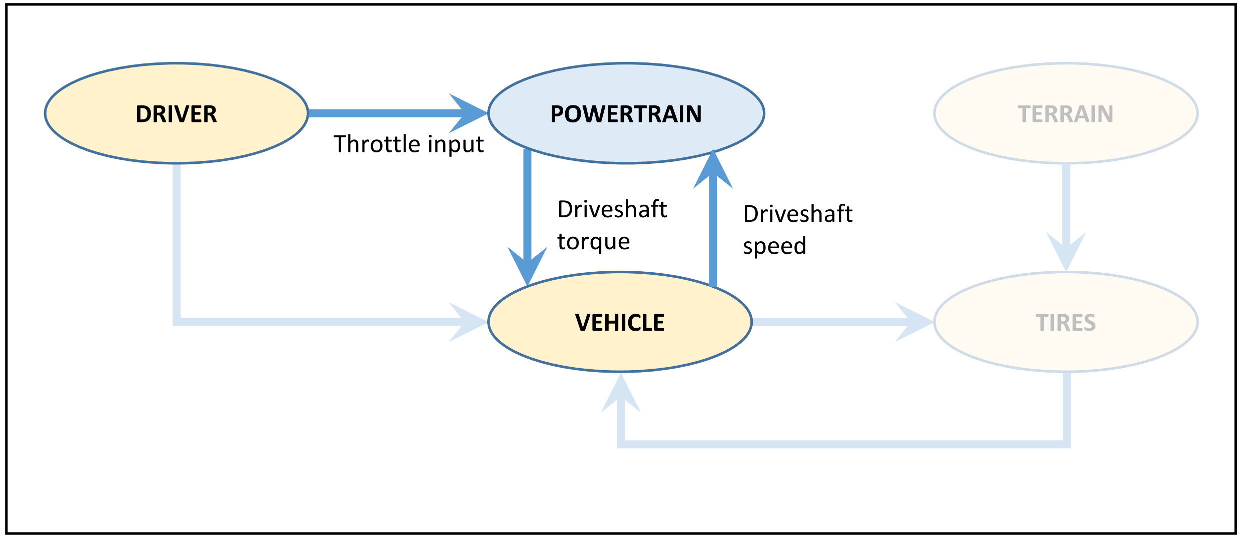
```
/// Construct using the specified Bezier curve.
ChPathFollowerDriver(ChVehicle& vehicle,             ///< associated vehicle
                     ChBezierCurve* path,            ///< Bezier curve with target path
                     const std::string& path_name,   ///< name of the path curve
                     double target_speed,            ///< constant target speed
                     bool isClosedPath = false       ///< Treat the path as a closed loop
                     );

/// Construct using JSON specification files.
/// The two files must contain specification for the path-follower steering controller
/// and the constant-speed controller, respectively.
ChPathFollowerDriver(ChVehicle& vehicle,                     ///< associated vehicle
                     const std::string& steering_filename,   ///< JSON file with steering controller specification
                     const std::string& speed_filename,      ///< JSON file with speed controller specification
                     ChBezierCurve* path,                    ///< Bezier curve with target path
                     const std::string& path_name,           ///< name of the path curve
                     double target_speed,                    ///< constant target speed
                     bool isClosedPath = false               ///< Treat the path as a closed loop
                     );
```

Powertrain Models

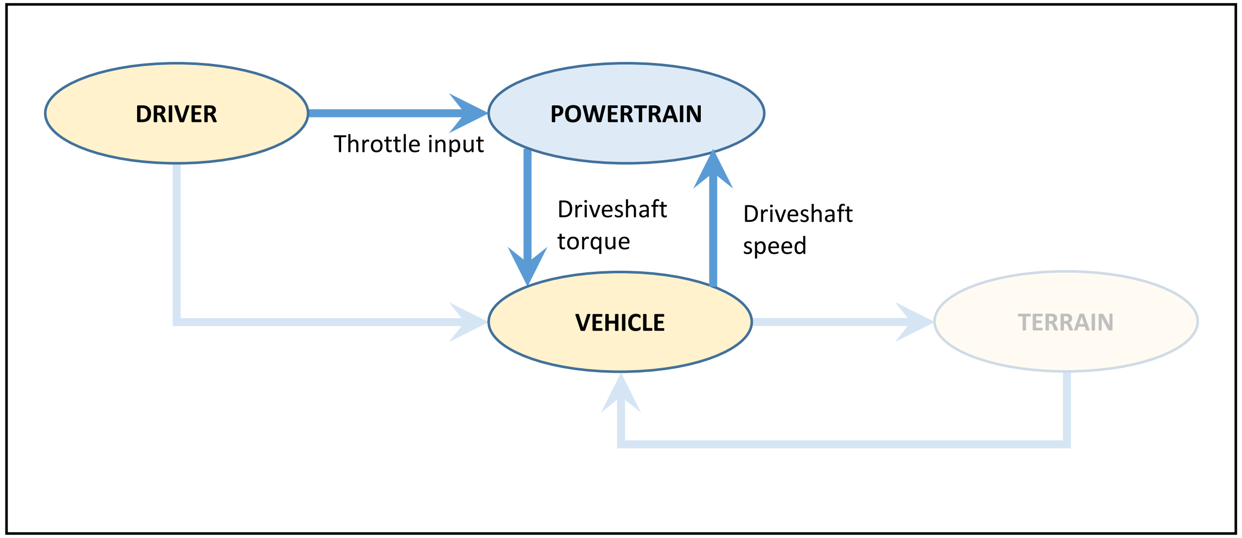
Simple Powertrain

Shafts Powertrain

Data flow



Wheeled vehicles



Tracked vehicles

ShaftsPowertrain model

- Uses Chrono ChShaft elements

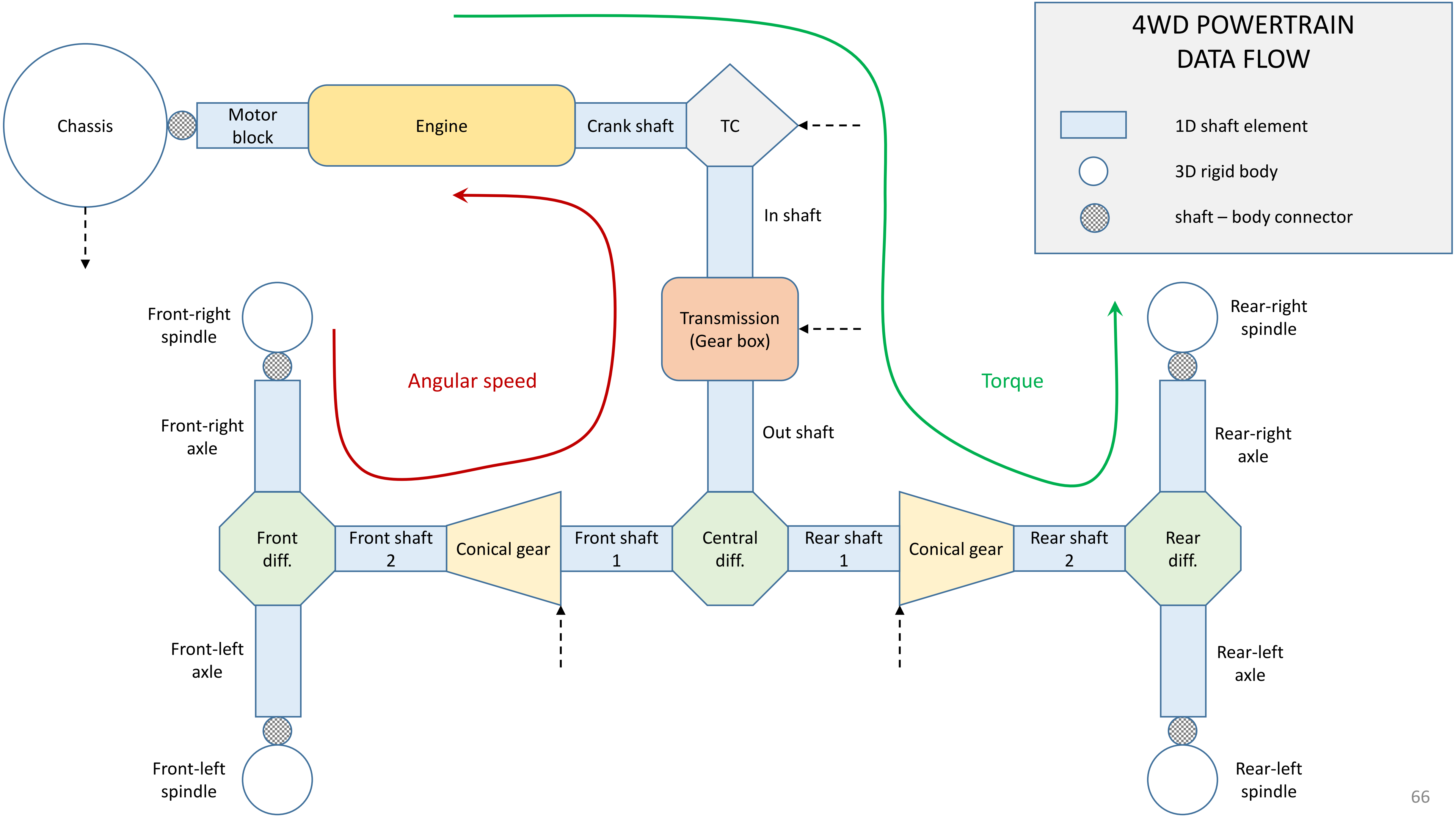
- Engine model based on speed-torque curves: $\tau_E = \tau_E(\omega_E)$

- Torque converter model uses two curves:
  - capacity factor curve: $K = K(R_\omega)$
  - torque ratio curve: $R_\tau = R_\tau(R_\omega)$

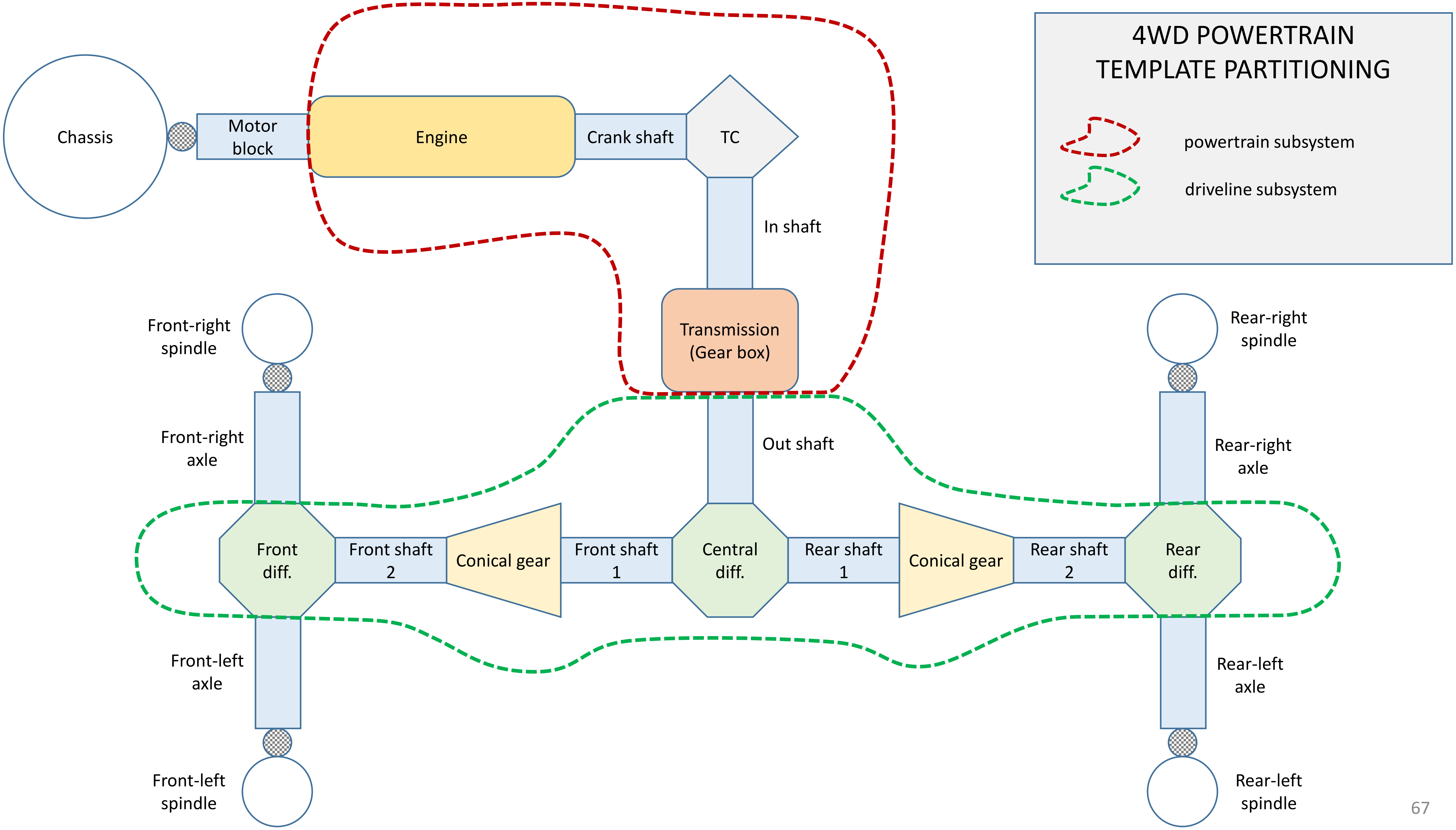
  where

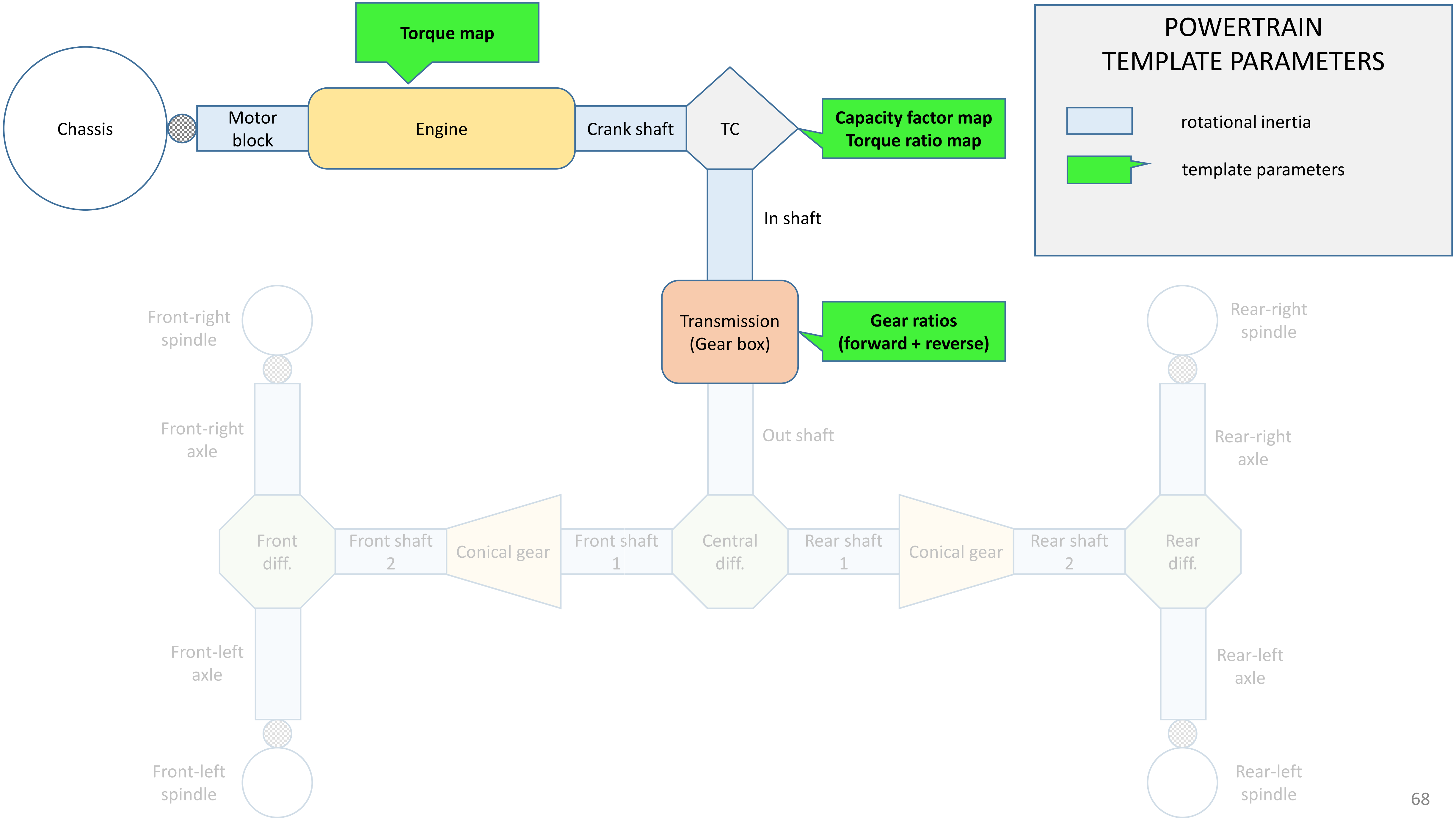
$R_\omega = \omega_T / \omega_I$	speed ratio (turbine – impeller)
$R_\tau = \tau_T / \tau_I$	torque ratio (turbine – impeller)
$K = \omega_I / \sqrt{\tau_I}$	capacity factor

- Transmission is a gear box, parameterized by a set of forward gear ratios and a single reverse gear ratio


4WD POWERTRAIN
DATA FLOW
1D shaft element
3D rigid body
shaft – body connector
Chassis
Motor block
Engine
Crank shaft
TC
In shaft
Transmission (Gear box)
Out shaft
Angular speed
Torque
Front-right spindle
Front-right axle
Front diff.
Front shaft 2
Conical gear
Front shaft 1
Central diff.
Rear shaft 1
Conical gear
Rear shaft 2
Rear diff.
Rear-right spindle
Rear-right axle
Front-left axle
Front-left spindle
Rear-left axle
Rear-left spindle

4WD POWERTRAIN TEMPLATE PARTITIONING




Torque map
Chassis
Motor block
Engine
Crank shaft
TC
Capacity factor map
Torque ratio map
In shaft
Transmission (Gear box)
Gear ratios (forward + reverse)
Out shaft
Front-right spindle
Front-right axle
Front diff.
Front shaft 2
Conical gear
Front shaft 1
Central diff.
Rear shaft 1
Conical gear
Rear shaft 2
Rear diff.
Front-left axle
Front-left spindle
Rear-right spindle
Rear-right axle
Rear-left axle
Rear-left spindle
POWERTRAIN TEMPLATE PARAMETERS
rotational inertia
template parameters

JSON specification file for ShaftsPowertrain

```
{
  "Name":                       "Generic Shafts Powertrain",
  "Type":                       "Powertrain",
  "Template":                   "ShaftsPowertrain",

  "Engine":
  {
    "Motor Block Inertia":      10.5,
    "Crankshaft Inertia":       1.1,
    "Torque Map":               [
                                   [ -100 , 300 ],
                                   [  800 , 382 ],
                                   [  900 , 490 ],
                                   [ 1000 , 579 ],
                                   [ 1100 , 650 ],
                                   [ 1200 , 706 ],
                                   [ 1300 , 746 ],
                                   [ 1400 , 774 ],
                                   [ 1500 , 789 ],
                                   [ 1600 , 793 ],
                                   [ 1700 , 788 ],
                                   [ 1800 , 774 ],
                                   [ 1900 , 754 ],
                                   [ 2000 , 728 ],
                                   [ 2100 , 697 ],
                                   [ 2200 , 664 ],
                                   [ 2300 , 628 ],
                                   [ 2400 , 593 ],
                                   [ 2500 , 558 ],
                                   [ 2700 ,-400 ]
                                ],
    "Losses Map":               [
                                   [  -50 ,  30 ],
                                   [    0 ,   0 ],
                                   [   50 , -30 ],
                                   [ 1000 , -50 ],
                                   [ 2000 , -70 ],
                                   [ 3000 , -90 ]
                                ]
  },
```

Subsystem type (string)

Template type (string)

Speed – motor torque curve

Speed – resistive torque curve

JSON specification file for ShaftsPowertrain

```
"Transmission":
 {
   "Input Shaft Inertia":   0.3,
   "Reverse Gear Ratio":    -0.1,
   "Forward Gear Ratios":   [0.2, 0.4, 0.8]
 },

 "Torque Converter":
 {
   "Capacity Factor Map":  [
                             [ 0.00 , 15.00 ],
                             [ 0.25 , 15.00 ],
                             [ 0.50 , 15.00 ],
                             [ 0.75 , 16.00 ],
                             [ 0.90 , 18.00 ],
                             [ 1.00 , 35.00 ]
                           ],
   "Torque Ratio Map":     [
                             [ 0.00 , 2.00 ],
                             [ 0.25 , 1.80 ],
                             [ 0.50 , 1.50 ],
                             [ 0.75 , 1.15 ],
                             [ 1.00 , 1.00 ]
                           ]
 }
}
```

Forward gear ratios (1^{st}, 2^{nd}, etc.)

Torque converter capacity factor curve
(capacity factor as function of speed ratio)

Torque converter torque ratio curve
(torque ratio as function of speed ratio)

SimplePowertrain model

- Very simple model:

$$\omega_m = \omega_s / g$$
$$\tau_m = throttle \cdot \tau_{max} \cdot (1 - \omega_m / \omega_{max})$$
$$\tau_s = \tau_m / g$$

- No torque converter, no transmission

JSON specification file for SimplePowertrain

```
{
  "Name":                     "Generic Simplified Powertrain",
  "Type":                     "Powertrain",
  "Template":                 "SimplePowertrain",

  "Forward Gear Ratio":       0.3,
  "Reverse Gear Ratio":       -0.3,
  "Maximum Engine Torque":    272.0,
  "Maximum Engine Speed":     2000
}
```

Subsystem type (string)

Template type (string)

Visualization

Runtime visualization with Irrlicht

Specifying visualization type

- Visualization type is controlled on a per-subsystem level
- Available settings: `NONE`, `PRIMITIVES`, `MESH` (if supported)
- ChVehicle and derived classes provide functions `Set***VisualizationType` which must be called after vehicle initialization

- Base vehicle subsystem visualization:

```
/// Set visualization mode for the chassis subsystem.
void SetChassisVisualizationType(VisualizationType vis);
```

Specifying visualization type

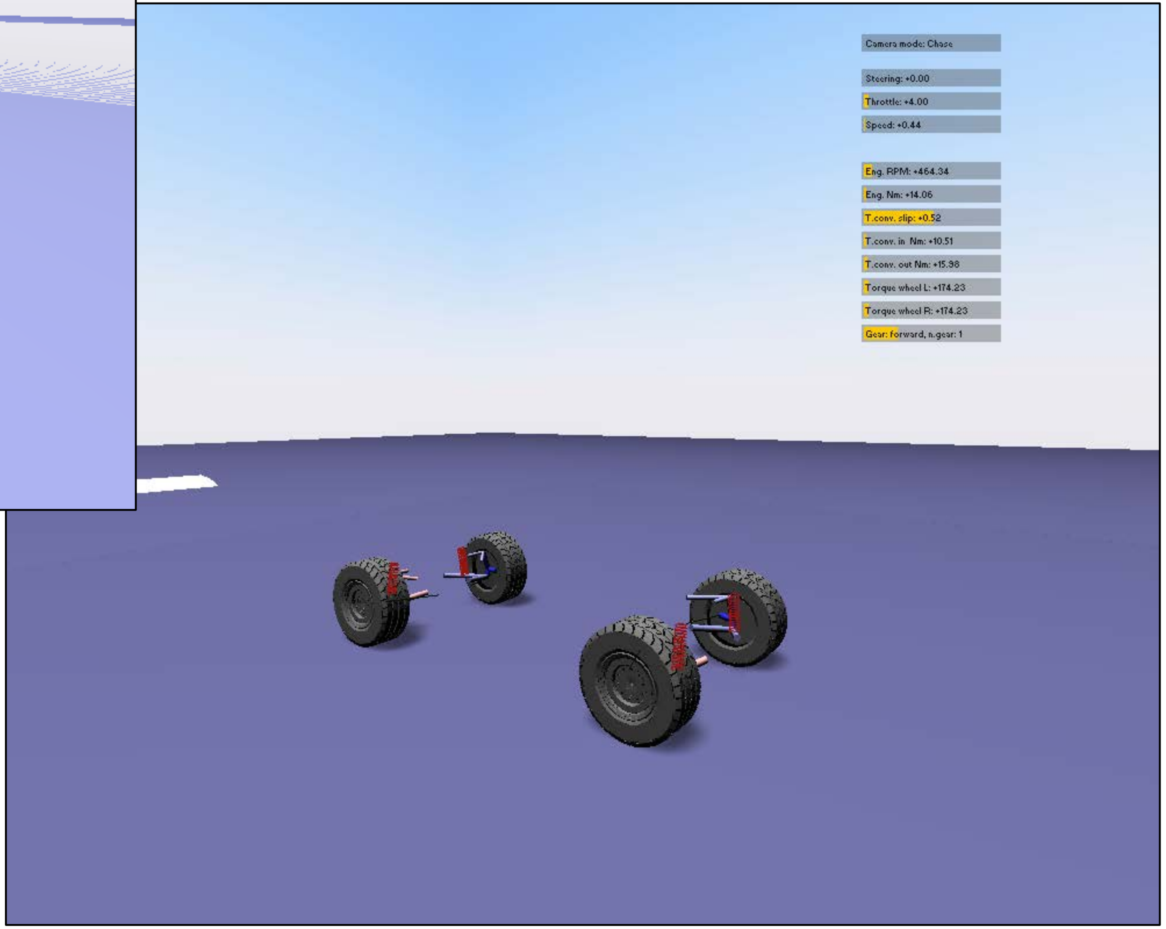
- Wheeled vehicle subsystem visualization:

```
/// Set visualization type for the suspension subsystems.
/// This function should be called only after vehicle initialization.
void SetSuspensionVisualizationType(VisualizationType vis);

/// Set visualization type for the steering subsystems.
/// This function should be called only after vehicle initialization.
void SetSteeringVisualizationType(VisualizationType vis);

/// Set visualization type for the wheel subsystems.
/// This function should be called only after vehicle initialization.
void SetWheelVisualizationType(VisualizationType vis);
```

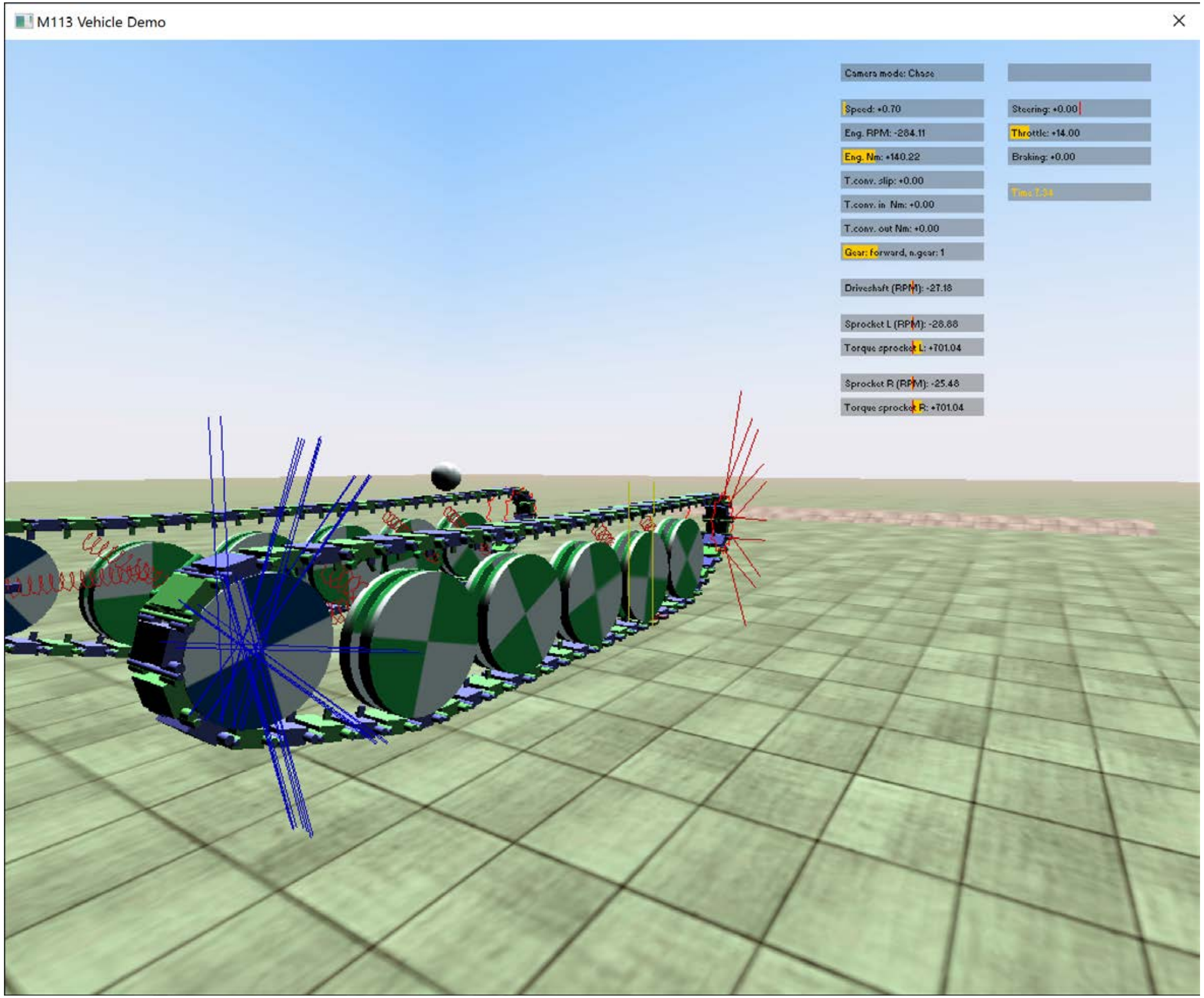
- Tracked vehicle subsystem visualization:

```
/// Set visualization type for the sprocket subsystem.
void SetSprocketVisualizationType(VisualizationType vis);

// Set visualization type for the idler subsystem.
void SetIdlerVisualizationType(VisualizationType vis);

/// Set visualization type for the suspension subsystems.
void SetRoadWheelAssemblyVisualizationType(VisualizationType vis);

/// Set visualization type for the track shoe subsystems.
void SetTrackShoeVisualizationType(VisualizationType vis);
```

Camera mode: Chase
Steering: +0.00
Throttle: +2.00
Speed: +0.16
Eng. RPM: +307.20
Eng. Nm: +6.74
T.conv. slip: +0.74
T.conv. in Nm: +4.60
T.conv. out Nm: +8.21
Torque wheel L: +97.31
Torque wheel R: +97.31
Gear: forward, n.gear: 1



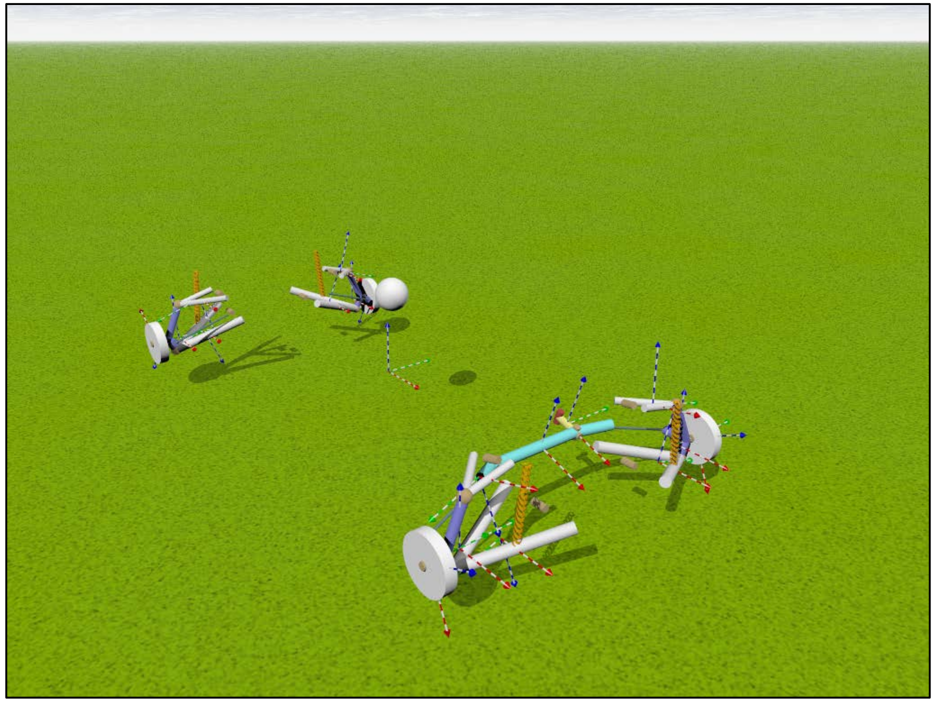
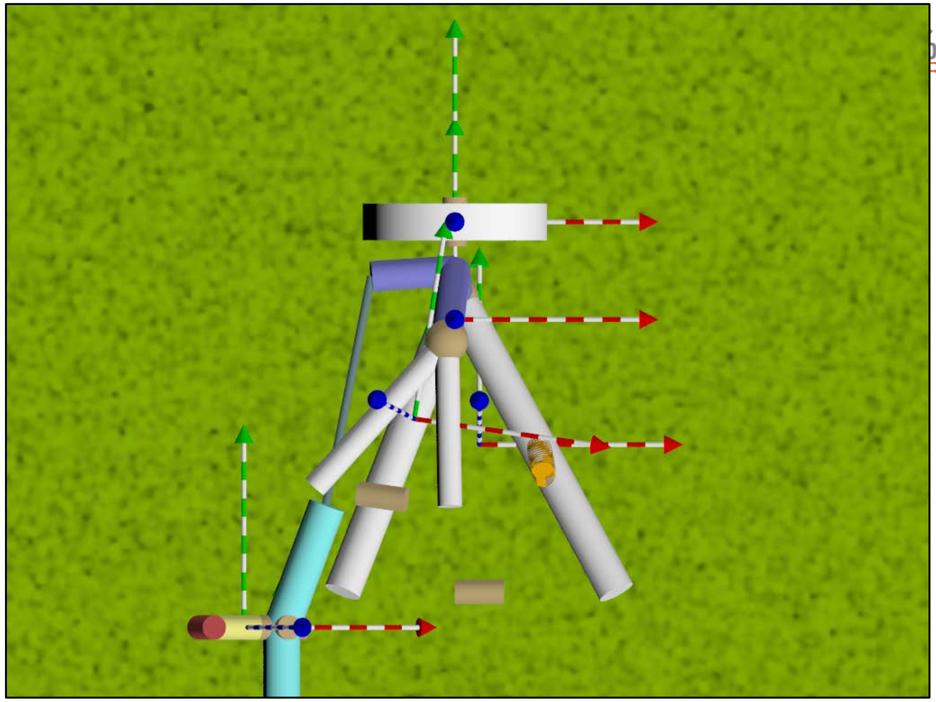
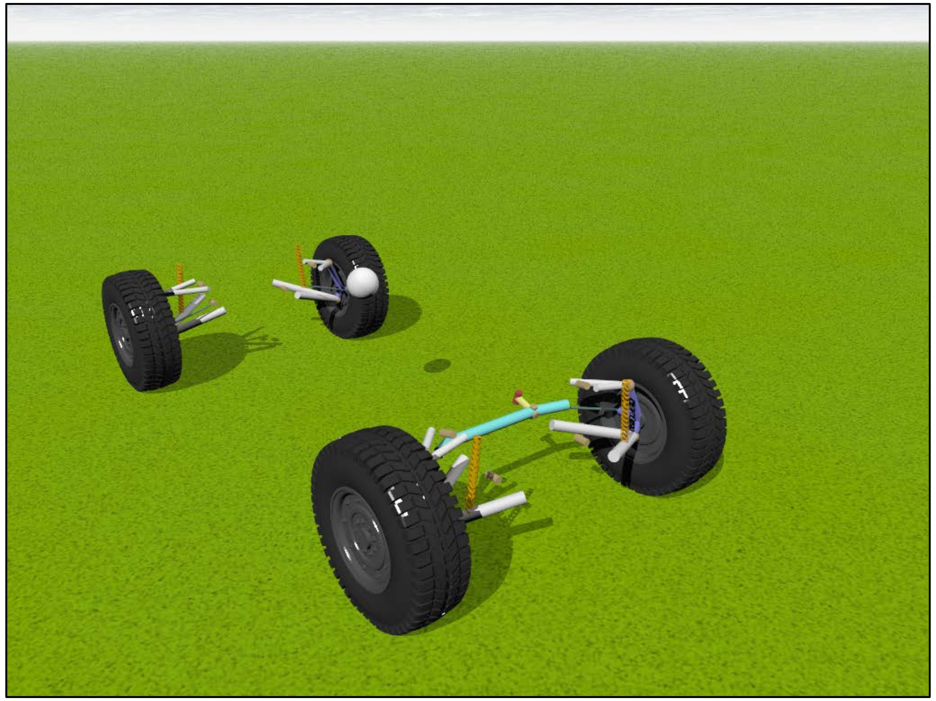
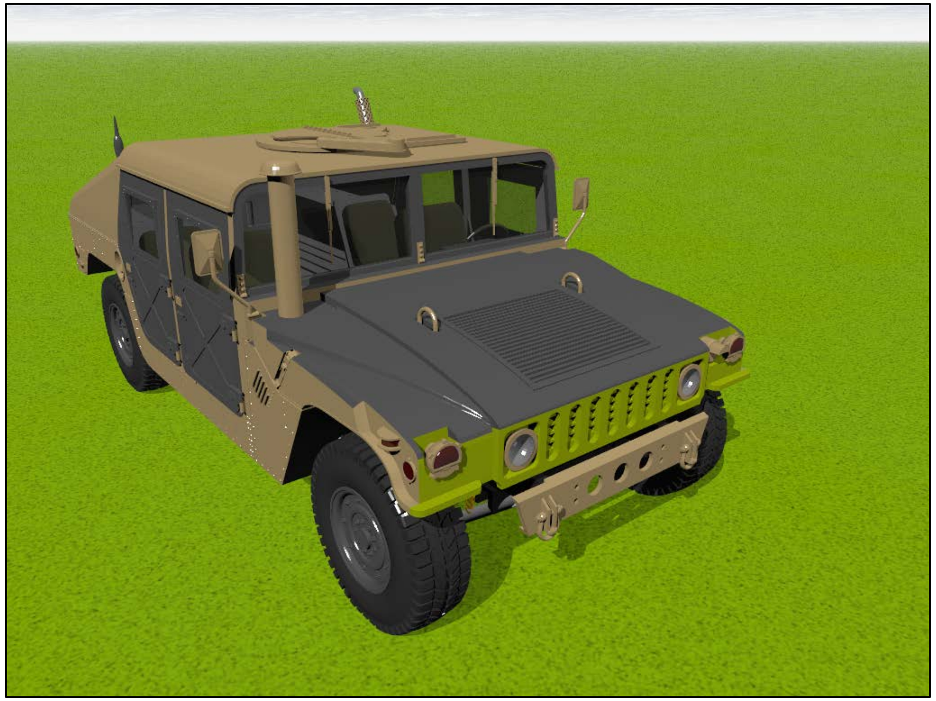
Camera mode: Chase
Steering: +0.00
Throttle: +4.00
Speed: +0.44
Eng. RPM: +464.34
Eng. Nm: +14.06
T.conv. slip: +0.52
T.conv. in Nm: +10.51
T.conv. out Nm: +15.98
Torque wheel L: +174.23
Torque wheel R: +174.23
Gear: forward, n.gear: 1



M113 Vehicle Demo
Camera mode: Chase
Speed: +0.70
Eng. RPM: -284.11
Eng. Nm: +140.22
T.conv. slip: +0.00
T.conv. in Nm: +0.00
T.conv. out Nm: +0.00
Gear: forward, n.gear: 1
Driveshaft (RPM): -27.18
Sprocket L (RPM): -28.88
Torque sprocket L: +701.04
Sprocket R (RPM): -25.48
Torque sprocket R: +701.04
Steering: +0.00
Throttle: +14.00
Braking: +0.00
Time 7.34

Visualization

Post-processing with POV-Ray

Preparing output data files – WriteShapesPovray

- Call this function at each simulation frame that needs to be post-processed
- Outputs a Comma Separated Value (CSV) file with specified name
- File contains information for
  - All bodies in the Chrono system (position and orientation)
  - All visualization assets (position, orientation, type, parameters)
  - Selected types of joints

```
// Write CSV output file for PovRay.
// Each line contains information about one visualization asset shape, as
// follows:
//    index, x, y, z, e0, e1, e2, e3, type, geometry
// where 'geometry' depends on 'type' (an enum).
CHApi
void WriteShapesPovray(ChSystem*           system,
                       const std::string& filename,
                       bool               body_info = true,
                       const std::string& delim = ",");
```

Preparing output data files – WriteMeshPovray

- Call this function once to generate a PovRay input file from a specified Waveform OBJ mesh file
- Quick and dirty alternative to using a more powerful tool (e.g., PoseRay – https://sites.google.com/site/poseray/)

```
// Write the triangular mesh from the specified OBJ file as a macro in a PovRay
// include file. The output file will be "[out_dir]/[mesh_name].inc". The mesh
// vertices will be tramsformed to the frame with specified offset and
// orientation.
CHApi
void WriteMeshPovray(const std::string&    obj_filename,
                     const std::string&    mesh_name,
                     const std::string&    out_dir,
                     const ChColor&        color = ChColor(0.4f, 0.4f, 0.4f),
                     const ChVector<>&     pos = ChVector<>(0, 0, 0),
                     const ChQuaternion<>& rot = ChQuaternion<>(1, 0, 0, 0));
```

Creating PovRay images – renderZ.pov script

- Generic script for (batch) processing of output files in the format generated by WriteShapesPovray
- Assumes all data is expressed in a right-hand frame with Z up (and performs all required transformations to PovRay's Y-up left-handed frames)
- User controls:
  - Render a single frame or a sequence of frames (batch processing)
  - Turn on/off rendering of body and object (asset) reference frames
  - Turn on/off rendering of objects (assets)
  - Turn on/off rendering of static objects (assets of bodies fixed to ground)
  - Turn on/off rendering of links (joints)
  - Camera location and "look-at" point (with a few presets)
  - Enable/disable shadows
  - Optionally render environment (ground and sky)