

Building a push-back simulator from Scratch : AEROPVS020

Building a push-back simulator from scratch using **vsTASKER** involves setting up a dynamic simulation using real-time Newtonian laws, articulated physics, and behavior trees. You can create a fully functional visual demonstrator with an empty database without relying on predefined components.

1. Requirements and Setup

To build the demonstrator from scratch, you will need the following tools and assets:

- **vsTASKER:** The core simulation software (version 6.2+).
- **Visual Engine:** OpenSceneGraph (requires some programming knowledge), VBS-IG, or integration with Unreal/Unigine.
- **Airport Terrain/Database:** Provided natively or sourced (e.g., from TrianGraphics).
- **3D Models:** Free models of an aircraft, push-back tug, and tow bar (or tow-bar-less connection).
- **Front End UI Builder:** Built-in visual HMI toolkit in vsTASKER to interact with the scenario.

2. Physical and Logical Implementation

The core mechanic of the push-back revolves around the tow bar attachment and the physics applied to the articulated parts.

- **Physics and Math:** Because you don't want to extrapolate trajectories, implement a high-frequency dynamic replica of applied forces (momentum, angles over time) using standard Newton's laws of motion.
- **Vehicle Origin Points:** In your 3D modeling software, make sure the pushback vehicle's origin (0,0,0) is positioned at the exact point where the tug connects to the aircraft's nose gear, rather than the center of the vehicle itself.

- **Behavior Trees and State Machines:** Instead of raw programming, you can visually design the behavior of the tug. Map out states such as *Connecting to Nosegear*, *Pushing Back*, *Turning*, and *Disconnecting* using vsTASKER's State Machine tools.

3. Scenario Execution

Once the environment is mapped, you can generate the license-free simulation engine.]

- Define paths and waypoints (e.g., pulling back from the gate, aligning on the taxiway, and releasing the tow bar).
- Set up specific operating conditions, such as pushing on an icy surface, different lighting (day/night), or handling tight corners (useful for complex layouts like an aircraft carrier deck).
- For more details on the exact parameters and step-by-step document guidelines, check out the official VirtualSim Push-back Portfolio.

Would you like to know more about:

- **Integrating vsTASKER** with graphic engines like Unreal/Unigine or VBS?
- **Using C++ generation** to refine the Newton physics for the tow bar?
- Setting up **specific training scenarios** (like day/night cycles or slippery aprons)?

Building a push-back simulator from Scratch

by VirtualSim

This proof of concept demonstrates the use of vsTASKER to simulate a pushback maneuver on an airport tarmac. The final intend is to proceed on an aircraft carrier where limited space and congestion requires caution and training (second video below).

The pushback or towing implies articulated parts physic based on simple Newton laws. Under a real-time simulation process which does not require trajectory extrapolation, a high frequency dynamic replica of applied forces, momentum, angles over time will be sufficient to copy the reality with some assumptions (no drift for example).

The process of starting the prototype from scratch is explained in this video and in the following document which describe in detail the full process.

Read the full project description

What was needed:

- OpenSceneGraph (engine and some programming knowledge)
- Airport database (courtesy of TrianGraphics)
- 3D models for the aircraft, the tug and tow bar
- vsTASKER with some knowledge of how to use it
- Some basic mathematics for Newton equations

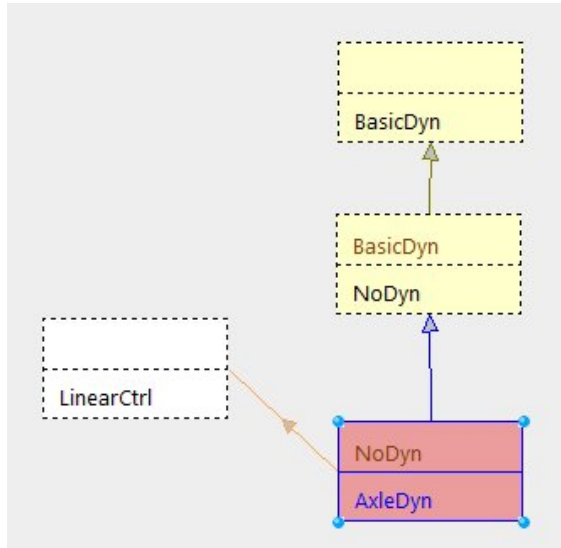


What we want to do

In this presentation, we will learn how to setup vsTASKER to build from scratch a pushback, tow and (multi-carriage) trailer simulator.

We'll start with an empty database and end up with a visual demonstrator using OpenSceneGraph and some free models available on Internet. vsTASKER used is v6.2. This way will give you an idea of the openness of the product and its possibilities as we will not use any canned devices or predefined components.

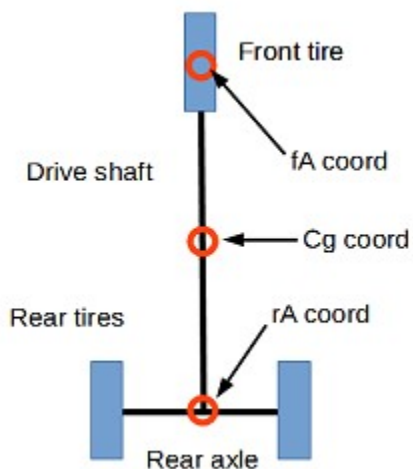
Preparation



In vsTASKER, most of the entities are represented by a 3 dimensional coordinates located at their centre of gravity, a velocity vector plus some attitude values (heading, pitch and roll). These variables are modified by equations of motion (entity dynamics) during runtime. Because we will run the new component at high frequency, we won't have to compute the trajectories with tons of equations but just let the vectors push and pull joints altogether. The result would normally conserve physic laws or stay close enough for the purpose of our simulation. We will see later that this assumption is correct.

For this exercise, we will need to create a new dynamic motion component for the chassis with simple equations based on Newton laws and vector forces. We will call this component **AxleDyn** as it will handle axles and tires. It will inherit from **NoDyn** component as we do not want to redefine all the pure virtual functions of **BasicDyn** class

Building the dynamic component



The AxleDyn component will have two axles separated by a drive shaft. We will only need to bother about the centre of each of these elements as they will be located under the carriage frame. At first, we will only set 3 tires. The front one connected to the steering wheel and the two rear tires, fixed on the rear axle, like the picture aside. Our chassis replica will only move the 3 points **fA**, **rA** and **cG**. fA and rA are connected by a **l** length shaft (see below) and the two rear tires are distanced by **w** meters (width of the chassis). The axles are always perpendicular to the shaft and this one gives the azimuth of the chassis.

We will need now to define the interface of our component. The more parameters we can let the user define, the more adaptable will it be when time to use the 3D will come. As vsTASKER automatically generates the user interface when data is preceded with `//&&`, we will do it for most of the variables.

Data Interface Section...

`// Data & Interface Section:`

`public:`

`float l; //&& LBL[Length of the drive shaft] DEF[2]`

`float w; //&& LBL[Width of the vehicle] DEF[1]`

`float lF; //&& LBL[Front Cantilevered] DEF[1.3]`

`float lR; //&& LBL[Rear Cantilevered] DEF[1]`

`float tL; //&& LBL[Tire length] DEF[0.6]`

`float tW; //&& LBL[Tire Width] DEF[0.15]`

`float max; //&& LBL[Maximum steering angle] DEF[90]`

This interface (above) will generate a configuration interface for the user to change the default values:

AxleDyn

Property	Value
Lenth of the drive shaft	20
Width of the vehicle	10
Front Cantilevered	13
Rear Cantilevered	6
Tire length	5
Tire Width	2.5
Maximum steering angle	90
Hooked to	
Rod Length	15

Refresh

☐ Auto Refresh

Default All

Rebuild All

Now, time to set the component variables used by the model.

Frame data definition...

LinearCtrl steer; //<! inertial controller of the wheel

Coord3D fT; //<! Front tire

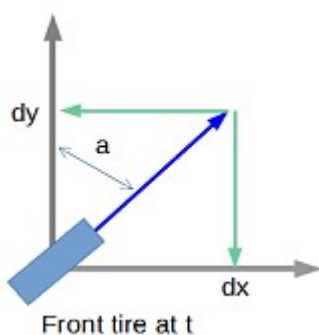
Coord3D rTl, rTr; //<! Rear left tire, Rear right tire

Coord3D fA; //<! Center of the front axle

Coord3D rA; //<! Center of the rear axle

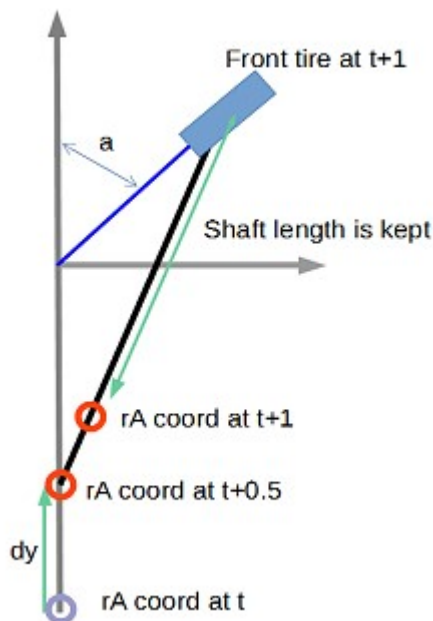
Coord3D Cg; //<! Center of the drive shaft (gravity center)

We can see that **steer** (the steering wheel) is based on a control loop component (LinearCtrl) used to soften the value changes and provide a kind of inertia (the current value will slightly change towards the target one over time).



Now, we will write the mathematical formulas for the model. For that, we need to remain simple by projecting vectors on each axis. Every cycle, the front axle will move according to the speed vector. At t_0 , according to the steering angle of the front tire (a) 2 projected distances are computed. One along the drive shaft axis (dy) and the second one on its perpendicular (dx). At t_1 , the front axle will move along the vector.

The rear axle will move in two steps. First move of (dx) along the shaft at t_0 and the second move along the shaft at t_1 , to keep the final length equal to (l). In reality, what happens is a combination of the two.



Runtime code called at every cycle...

```
steer.process();
```

```
// we get the distance travelled during a cycle
```

```
float d = getSpeed()*deltaT();
```

```
// We project on the orthonormal (Y along the drive shaft)
```

```
// the distance travelled on both axis
```

```
float dx = d*sin(steer+getHeading());
```

```
float dy = d*cos(steer+getHeading());
```

```

float ca = rA.azimuthTo(fA); // current azimuth of the drive shaft

// We move the rear axle center along the drive shaft, with the
// traveled value computed above (Y)
rA = rA.getNewPosAt(ca, 0, dy);

// The front axle center is now repositioned according the
// the speed vector (we assume there is no drift)
fA = fA.getNewPosAt(steer+getHeading(), 0, d);

float nl = rA.distanceTo(fA); // new distance between front and rear axle
float dl = l/nl; // ratio between what it is and what it should be

// We now slide the rear axle position along the drive shaft to keep the length
rA = fA.getNewPosAlong(rA, dl);

// update center of frame and heading
Cg = rA.getNewPosAlong(fA, 0.5);
pos->set(Cg); // update the entity position
setHeading(ca); // update the entity heading
updateTires(ca); // update position of all tires

```

Testing on OpenGL

Once the mathematics has been translated into code, we need to check the result on a 2D display. vsTASKER provides this capability out of the box. We just need to attach an OpenGL viewer and write the OpenGL code to display the frame and the tires shapes on the output window.

```
void AxleDyn::drawOn(OpenGLLib* ogl) ...

// *****

void AxleDyn::drawOn(OpenGLLib* ogl)

{

    // draw the vehicle

    float sf = scenario->terrain.db->scaleFactor();

    // draw the front tire

    ogl->setColor(clBlack);

    Coord2D sF = scenario->terrain.db->posToScreen(fT);

    ogl->drawRectangle(sF.x, sF.y, tW*sf, tL*sf, steer+getHeading());

    // draw the rear tires

    Coord2D sRl = scenario->terrain.db->posToScreen(rTl);

    Coord2D sRr = scenario->terrain.db->posToScreen(rTr);

    ogl->drawRectangle(sRl.x, sRl.y, tW*sf, tL*sf, getHeading());

    ogl->drawRectangle(sRr.x, sRr.y, tW*sf, tL*sf, getHeading());

    // draw the drive shaft

    Coord2D sfA = scenario->terrain.db->posToScreen(fA);

    Coord2D srA = scenario->terrain.db->posToScreen(rA);

    ogl->setColor(clGray);

    ogl->setLine(3);

    ogl->drawLine(srA,sfA);
```

```
// draw the rear axle
```

```
ogl->setLine(3);
```

```
ogl->drawLine(sRl,sRr);
```

```
// draw the frame
```

```
float a = rA.azimuthTo(fA);
```

```
Coord3D sC = scenario->terrain.db->posToScreen(Cg);
```

```
ogl->drawRectangle(sC, w*sf, getLength()*sf, a);
```

To make the testing more interesting, we can add some colored ground traces with simple lines added at the end of the runtime code:

Data needed for the ground traces...

```
Coord2D *f = new Coord2D;
```

```
Coord2D *rl = new Coord2D;
```

```
Coord2D *rr = new Coord2D;
```

```
f->set(scenario->terrain.db->posToScreen(fT));
```

```
rl->set(scenario->terrain.db->posToScreen(rTl));
```

```
rr->set(scenario->terrain.db->posToScreen(rTr));
```

```
pF.addElem(f); // pF declared as Array
```

```
pRl.addElem(rl); // pRl declared as Array
```

```
pRr.addElem(rr); // pRr declared as Array
```

And to display it, just add the following code to the function drawOn():

Ground traces draw code...

```
// draw the tire paths
```

```
ogl->setColor(clRed);
```

```
ogl->drawLines(pF);
```

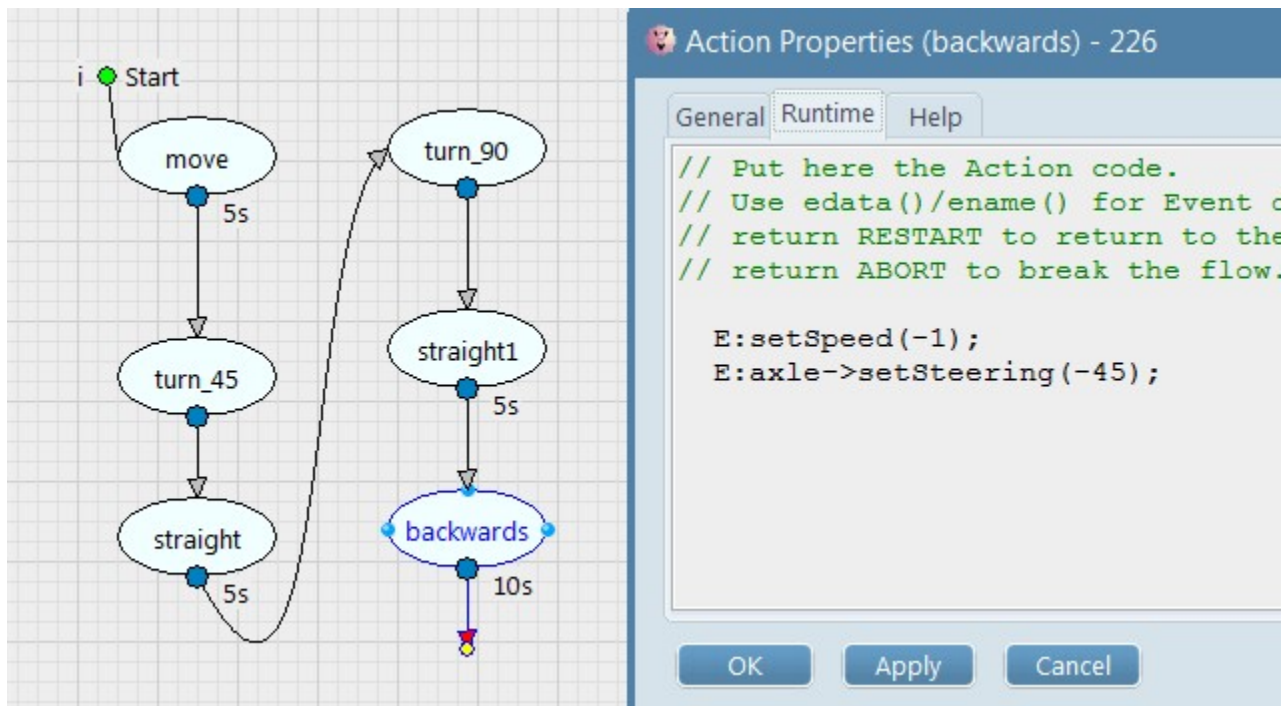
```
ogl->setColor(clGreen);
```

```
ogl->drawLines(pRl);
```

```
ogl->setColor(cIBlue);
```

```
ogl->drawLines(pRr);
```

In order to maneuver the vehicle, we define a logic made of successive **Actions** which controls the speed and the steering wheel or the tractor (direct call of the AxleDyn API functions):



Giving the following result (click on the video below):

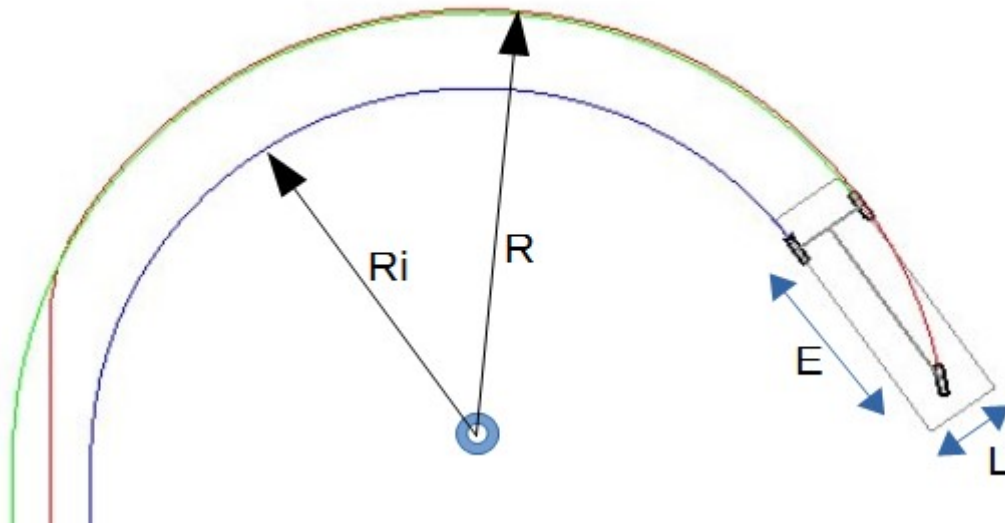
Video Player

00:00

00:23

Confronting the data

To insure that our dynamic is physically correct, we need to measure the turn radius of each tire for a given steering angle, and compare the data with the theoretical values. For that, we will store the position of each tire during a full circle (to get the circumference) then deduct the radius by dividing by 2π .



With **L: 2.5 m**, **E: 6.2 m**, steering **angle: 10°**, our simulation gives: **R: 39.4**, **Ri: 37.6**

When confronted with theoretical data, we can see we are very good indeed:

$$R = \frac{E}{\sin \alpha} = 39.416$$

$$R_i = \sqrt{\left(\frac{E}{\sin \alpha}\right)^2 - E^2 + P_r^2 - E_r^2} - \frac{L}{2} = 37.675$$

Adding trailers

Now, it is time to test adding a trailer on the rear hook of our vehicle.

For that, we need to extend the AxleDyn component with the name of the tractor (entity) to hook the trailer to. Then, we will add a trailer (entity) into the scenario, attach an AxleDyn component to it, setup parameters to hook to the current tractor and let the chassis (component) be automatically driver by the tractor (hook).

For that, let extend the component the following way:

Axle component extension...

```
int axles;    //&& LBL[Number of axles] DEF[2] LIST[1=1, 2=2]
```

```
char tug_name[NS]; //&& LBL[Hooked to] TYPE[Entity]
```

```
float tow_bar;  //&& LBL[Rod Length] DEF[0]
```

```
AxleDyn* tug;   // in case this axle is towed by a tractor
```

```
/// Return the position of the current rear hook
```

```
Coord2D AxleDyn::getRearHook()
```

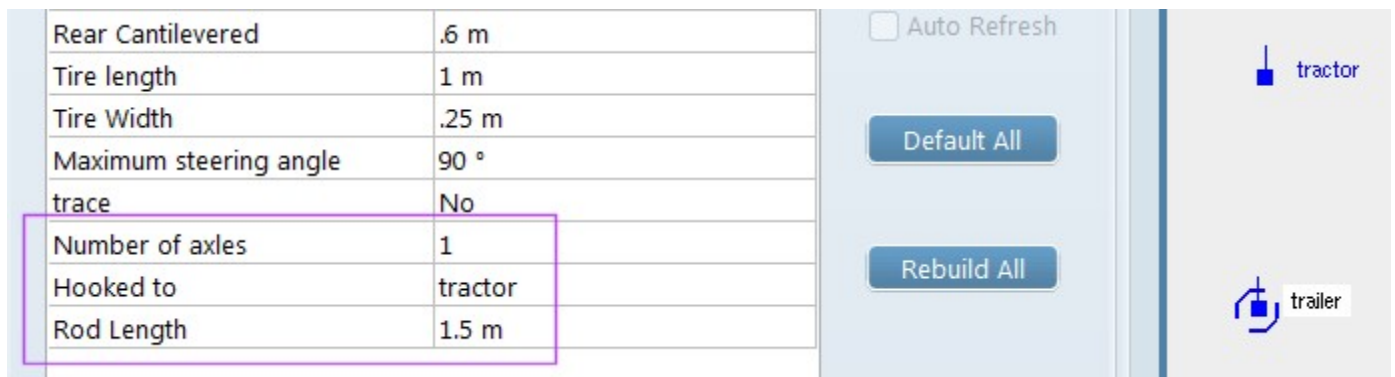
```
{
```

```
    float a = fA.azimuthTo(rA);
```

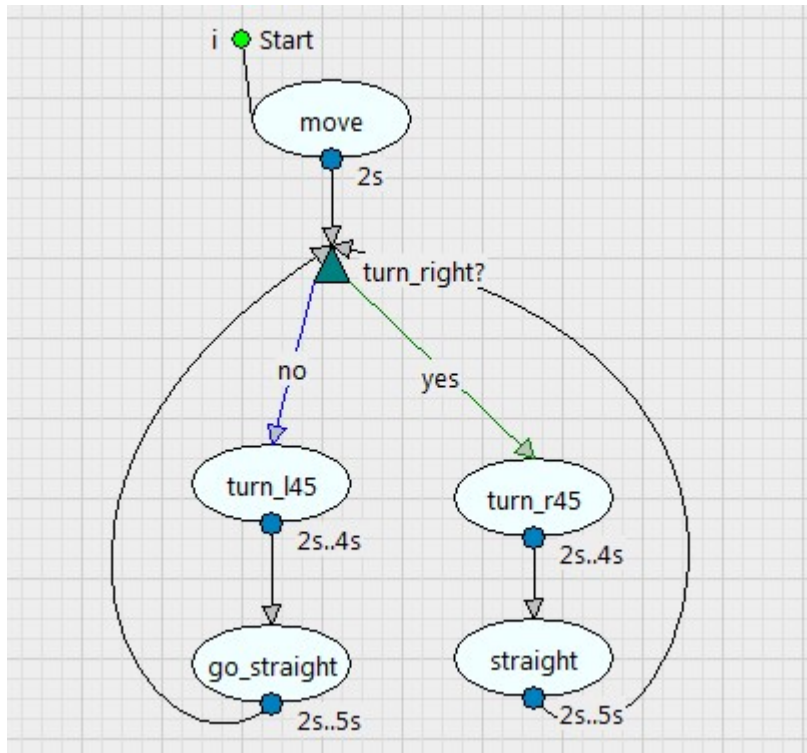
```
    return rA.getNewPosAt(a, 0, lR);
```

```
}
```

Now, time to add two trailers, attached together up to the tractor vehicle (see image below). We can also notice that we have specified each trailer has only one axle (the rear one) and a tow bar of 1.5 meter long.



Now, we define a fairly simple logic to make the tractor wander randomly in the area:



and we obtain the following result (click on the video below):

Video Player

00:00

00:47

Checking with a 2 axles trailer

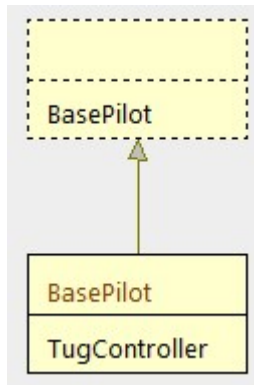
Now, let's see how the dynamic model works with a two axles trailer. We decide that the tow bar is no more directing the front wheel but let this one automatically adjust itself according to the towing direction. Here is the result, proving that decomposition of complex interactions using simple concepts produces a realistic outcome (tow bar of 4 meters) :

Video Player

00:00

00:32

Pushback using a tow bar



At this time, we are close enough to reach the purpose of this exercise: pushback (of an aircraft) using a tug. In real life, there is two kind of pushback methods: using a tow-bar and without.

Because pushing is much sensitive than pulling, we will use a gamepad to control the speed and the steering of the tug. Adding such a component is straightforward as the basic controller is already available in the vsTASKER built-in component library. Writing the interface class between the controller and the tug will be easy.

First thing is to inherit from the base component which reads the gamepad values and connect them to the AxleDyn API.

Controlling the steering wheel...

```
if (ax < -0.5) axle->incSteering(-0.5);
```

```
if (ax > 0.5) axle->incSteering(0.5);
```

```
if (az > 0.5) axle->incSpeed(0.01);
```

```
if (az < -0.5) axle->incSpeed(-0.01);
```

```
if (but == 2) axle->setSpeed(0);
```

To help driving the tug, we need a visual indicator showing the orientation of the front wheel:

Visual heading indicator

```
ogl->setColor(clOrange);
```

```
Coord2D Ai = fA.getNewPosAt(steer+getHeading(), 0, 5);
```

```
Coord2D sAi = scenario->terrain.db->posToScreen(Ai);
```

```
ogl->drawLine(sfA, sAi); // sFa = Front Axle on screen
```

Now, if we want to improve the simulation by making sure the front wheel of the aircraft stays in the angle limits, let's check them and update the display to turn the color to red when tow bar is broken.

Checking wheel limits...

```
// check the limit of the steering tire
```

```
if (fabs(RAD2DEG(steer)) > max) error = true;
```

```
...
```

```
// draw the tow bar (sHk is Coord2D private member of the class)
```

```
if (tow_bar && tug) {
```

```
    ogl->setLine(2);
```

```
    if (error) ogl->setColor(clRed);
```

```
    else {
```

```
        ogl->setColor(clBlack);
```

```
        sHk = scenario->terrain.db->posToScreen(tug->getHook());
```

```
    }
```

```
    ogl->drawLine(sfA, sHk);
```

```
}
```

Giving the following result (click on the video below):

Video Player

00:00

01:21

Setting OSG airport and 3D models

We are getting closer to the final result. We need now to use OSG to make the trainer more realistic. This part is the most challenging as combining 2D OpenGL algorithms with 3D physical models is always touchy. Happily, vsTASKER provides some nice tools and environment to ease this process.

First, we download a demo terrain database (courtesy of TrianGraphics) and we dropped on it some 3D models downloaded from TurboSquid website. The video below explains the process:

Video Player

00:00

01:41

We now need to adjust the chassis frame of our AxleDyn model to the dimensions and tire locations of both aircraft and tug vehicles. For that, it is enough to create some components tailored to display on the OSG output visual cues (location of the AxleDyn tires, vertical lines, etc) like we did previously on the test OpenGL window. Each component will be attached to their corresponding entity:



OSG frame cues data...

public:

```
float dx;    //&&
```

```
float dy;    //&&
```

```
int  display; //&& LIST[No=0,Yes=1] DEF[No]
```

protected:

```
osg::ref_ptr frame;
```

```
osg::ref_ptr fA, rRr, rRl;
```

```
void AxleOsg::setupFrame()...
```

```
void AxleOsg::setupFrame()
```

```
{
```

```
    if (!display) return;
```

```
frame = new osg::PositionAttitudeTransform();

// front tire

fA = new osg::PositionAttitudeTransform();

Coord3D p = axle->frontTireDt();

osg::Vec3 posA(dx+p.x, dy+p.y, p.z);

fA->setPosition(posA);

fA->addChild(vtOsg::createBoxGeode(.5,.5,.5, osg::Vec4(1,0,0,1)));


// rear left tire

rRl = new osg::PositionAttitudeTransform();

p = axle->rearTireLeftDt();

osg::Vec3 posL(dx+p.x, dy+p.y, p.z);

rRl->setPosition(posL);

rRl->addChild(vtOsg::createBoxGeode(.5,.5,.5, osg::Vec4(1,0,0,1)));


// rear right tire

rRr = new osg::PositionAttitudeTransform();

p = axle->rearTireRightDt();

osg::Vec3 posR(dx+p.x, dy+p.y, p.z);

rRr->setPosition(posR);

rRr->addChild(vtOsg::createBoxGeode(.5,.5,.5, osg::Vec4(1,0,0,1)));


frame->addChild(fA);

frame->addChild(rRl);
```

```
frame->addChild(rRr);
```

```
osg_root->models->addChild(frame);
```

```
}
```

We are now able to perfectly match the dimensions of our AxleDyn model with the real dimensions of the 3D object it carries (aircraft, tow bar and tug) which are sometimes unknown or because the centre of reference of some 3D models are not always set in the middle.

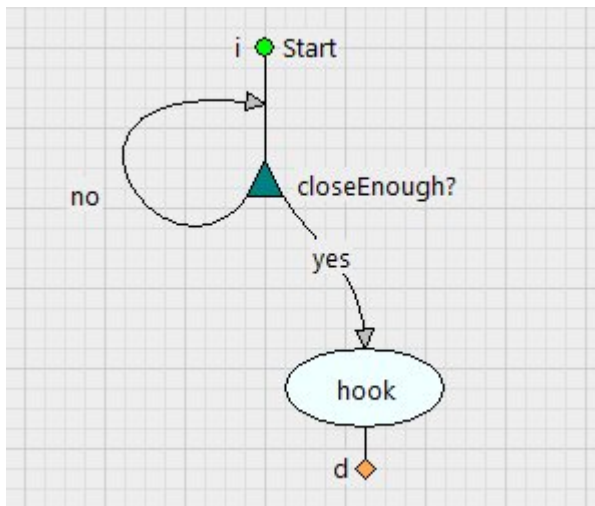
Once the camera is in place, changing the settings in vsTASKER, using the generated parameter panels allows visual adjustment, like shown in the following video:

Video Player

00:00

00:55

Finally, remains the tug automatic hooking of the tow bar to start the pushing. For that, we write a logic which computes the distance between the tug front hook and the tow bar hook in order to attach them as soon as below a given distance. It will be up to the trainee to drive (using the gamepad) the tug vehicle towards the aircraft tow bar.



Defining the variables...

```
AxleDyn* tugA, aircraftA;
```

```
Entity* bar = S:findEntity("bar");
```

```
tugA = E:findAxleDyn();
```

```
aircraftA = S:findEntity("aircraft")->findAxleDyn();
```

Checking the hook distance...

```
// compute the position of the extremity of the tow bar
```

```
WCoord bar_hook = L:bar->pos.getNewPosAt(L:bar->getHeading(),0,L:aircraftA->tow_bar/2);
```

```
WCoord tug_hook(L:tugA->getHook());
```

```
// compute the distance between the extremity of the tow bar and the tractor hook
```

```
float d = bar_hook.distanceTo(tug_hook);
```

```
if (d < L:gluedist) return YES;
```

```
return NO;
```

Engaging the hook...

```
// stop the tractor
```

```
L:tugA->setSpeed(0);
```

```
// hook it
```

```
L:aircraftA->setTractor(L:tugA->entity);
```

Final results

In the following two videos, you will see the final result of this exercise. First one is the pushback (with one try breaking the tow bar).

Video Player

00:00

02:42

The second video is pulling the aircraft (much easier process). To perform that, we just changed the tug hook setting from **Front** to **Rear** in the parameters window and we let the magic.

Video Player

00:00

01:57

Couple of days of work for one guy with a normal knowledge of vsTASKER (including OSG and writing this document) were necessary to carry out this prototype. If you are convinced, try our software yourself and see how it can help you with your ongoing or future projects.



OS & HARDWARE SUPPORT

INDUSTRY APPLICATION

Note: ** Specifications are copied from manufacturer data. Please contact before buying.