Этот урок будет довольно большим. Здесь мы постараемся сделать простой автомобильный симулятор. Хочется отметить что программ, в которых можно создавать нечто подобное (и даже лучше), довольно много. Из бесплатных и доступных можно назвать Blitz 3D и 3D Rad. В Blitz 3D главное неудобство заключается в отсутствии 3D редактора и возможности просмотра в режиме проектирования. И конечно, всё буквально приходится программировать на Basic подобном языке. А в остальном Blitz – хорошая штука и работает очень быстро, используя при этом DirectX 7. 3D Rad тоже использует DirectX. Но, в отличии от Blitz 3D, имеет просмотр в режиме проектирования и специально «заточен» под автомобильные симуляторы. Правда на нём можно делать и FPS и «леталки», но возникает огромная головная боль с импортом моделей. И ещё, 3D Rad довольно требователен к ресурсам и на слабой машине может сильно тормозить. Да и скриптовой язык у него (хоть и похож чем-то на Си) не имеет аналогов. Python, на мой взгляд, и проще и удобнее. Хотя в Blender нет обработчика ошибок. Поэтому, если вы не дописали запятую в строке, он вам ничего не скажет – просто не будет работать скрипт. Нужно быть очень внимательным.

Вывод прост – Blender удобнее и практичнее всего сказанного выше. Нет готовой модели – не беда! Мы можем её создать ☺ . Скорость работы BGE приемлема в простых проектах даже на слабых машинах. А то, что он использует Open GL, является большим плюсом. Ведь при этом отпадает проблема переносимости игры между разными операционными системами.

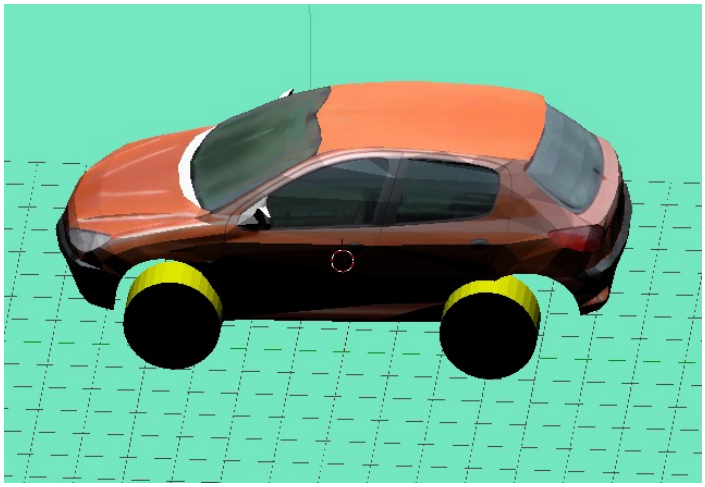
В разделе «Готовые проекты» я выложил заготовку к уроку. В архиве есть папка с текстурированной низкополигональной моделью автомобиля и звуками. Там же лежит сам файл проекта и .exe игры.

И так, скачиваем архив и берём из него папку Car_Blend2. Ложим папку Car_Blend2 рядом с проектом из урока 17.Теперь откроем проект урока с заготовкой автомобиля. Что нам мешает прикрепить готовую модель к нашей машинке? Ничего. Импортируем модель:

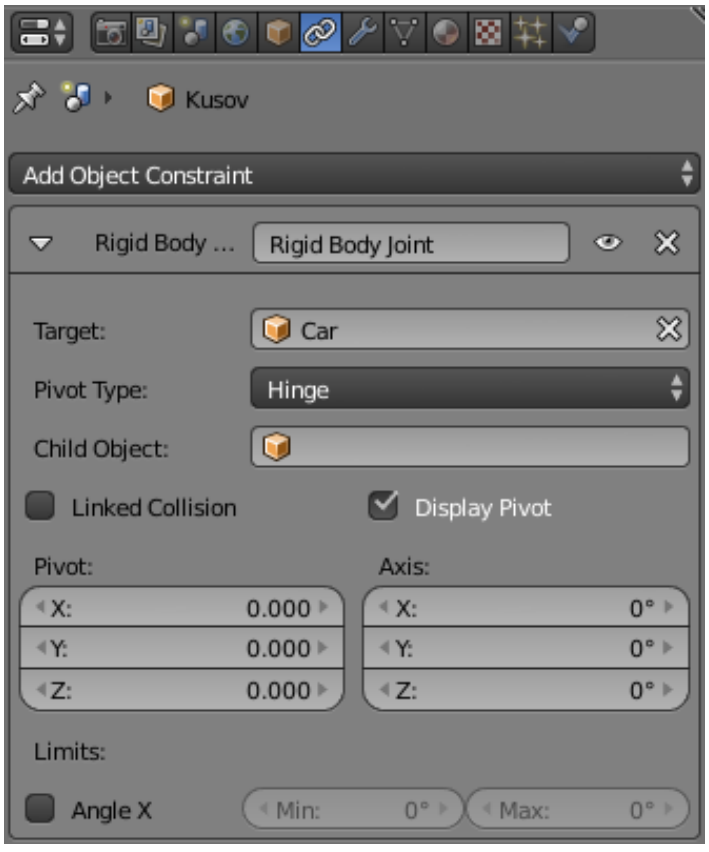
File->Import->3D Studio (.3ds)

Если у нас включено отображение текстур, то модель предстанет во всей красе:

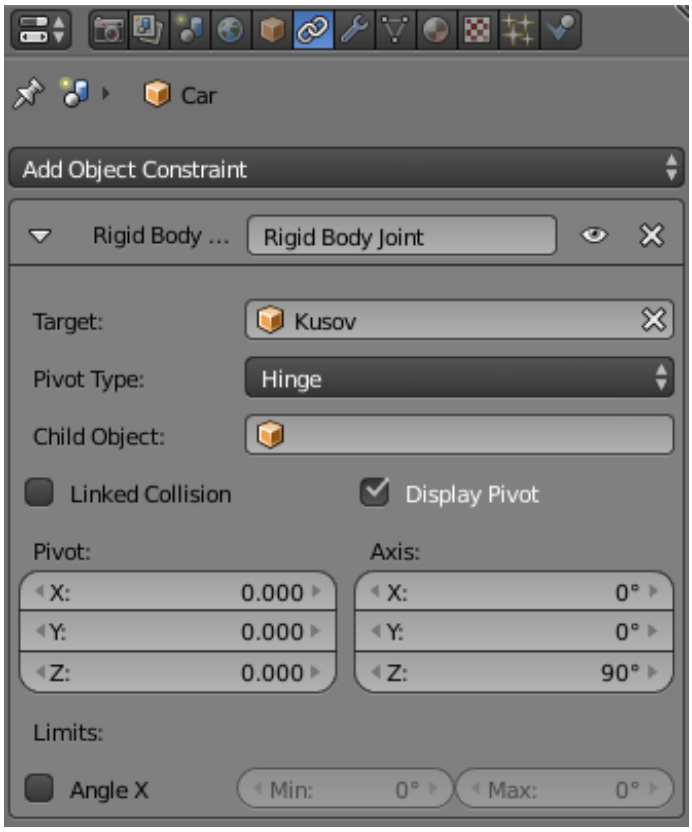
Нам нужно лишь разместить модель над колёсами и масштабировать её (лат. «S») таким образом, чтобы она аккуратно легла на колёса. Но предварительно в режиме редактирования опустим до уровня оси колёс корпус машинки. И ещё несколько изменений. В камере Focal Length уменьшим до 30. Свет поднимем повыше и добавим Energy до 2.5 . Выделим «землю» и разобъём её ещё раза три командой Subdivide. Тогда колёса перестанут утопать в грунте на склонах. В физике корпуса увеличим массу до 250. В Game Logic в скрипте управления (Powertrain_Standart) уменьшим ускорение (gasPower) до 250. Если, после масштабирования модели, колёса не совпадают с арками, то измените их расположение в скрипте CarSetup_Standart:



Теперь выделим модель и переименуем её в Kusov. Осталось только прикрепить её к плоскому корпусу машины двумя констрейнами. При выделенной модели создаём первый констрейн (в дальнейшем «Ограничение»):



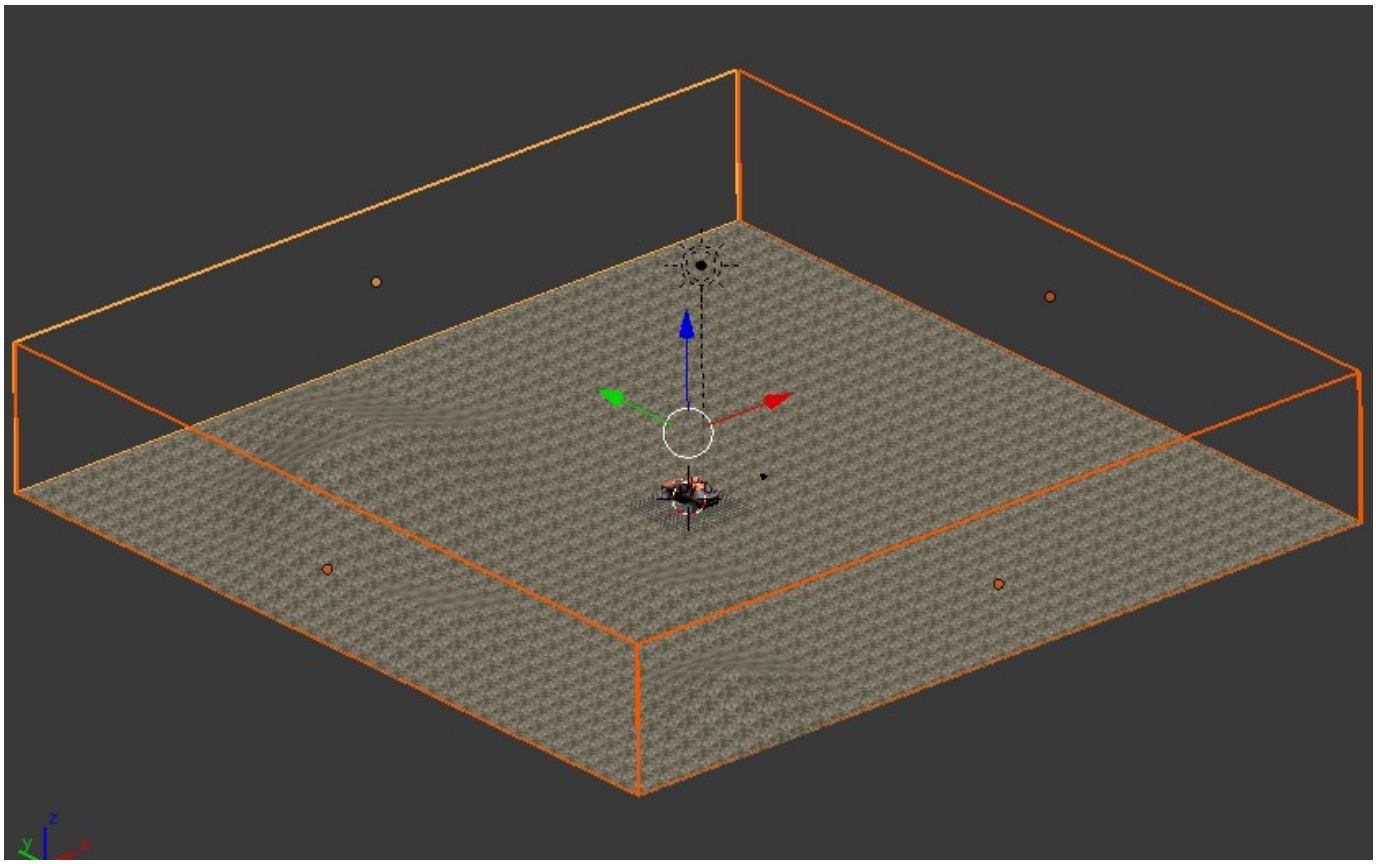
У него ось X расположена поперёк корпуса. Теперь выделяем корпус Car и создаём ещё одно ограничение. Только ось X разворачиваем по Z на 90 градусов. Она должна расположиться вдоль машины:



Снова выделим модель и настроим физику (Rigid Body и Collision). А в физике корпуса машины добавим галочку Ghost. Применим к модели Smooth (сглаживание) и готово:

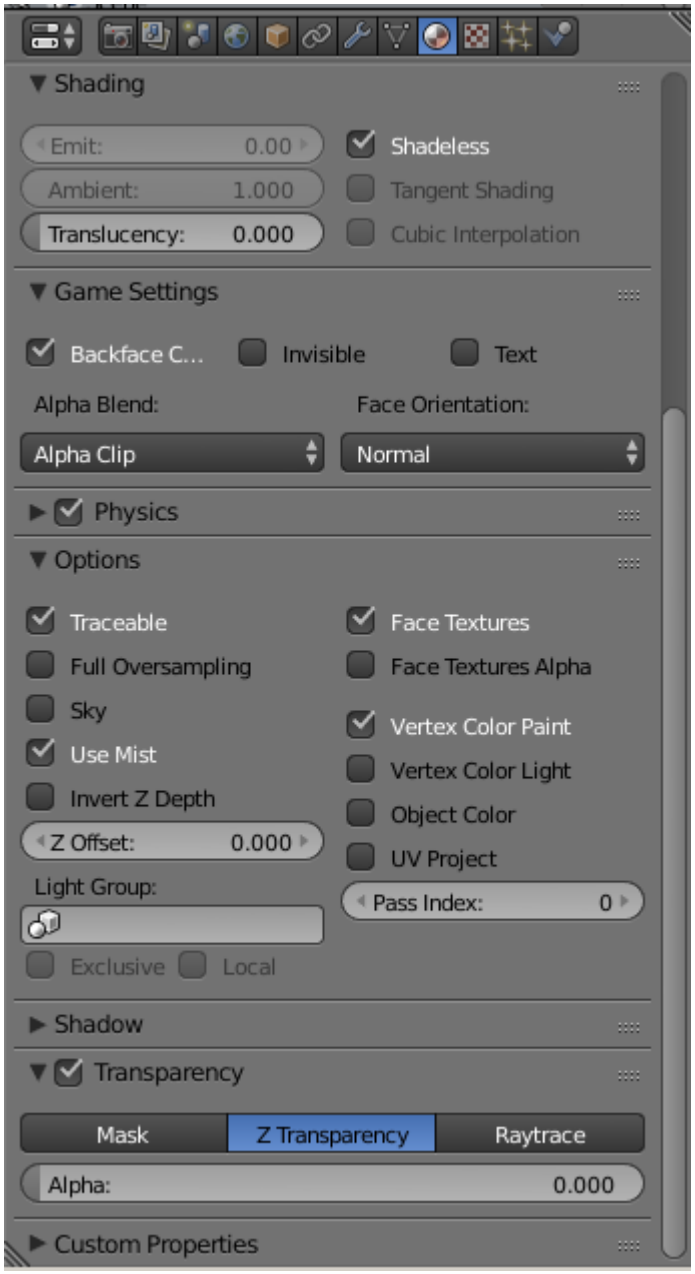
В папке Car_Blend лежат две текстуры для колеса. Там же лежит текстура камня для «земли». С этим вы должны справится сами, по опыту предыдущих уроков.

Кстати, в настройках скрипта CarSetup_Standart можно сделать мягче амортизаторы. Попробуйте поиграть значениями в строках # set suspension damping. А чтобы машина не падала в «пропасть», давайте окружим нашу «землю» прозрачными панелями. Расставим по краям планы с прозрачным материалом Stena:



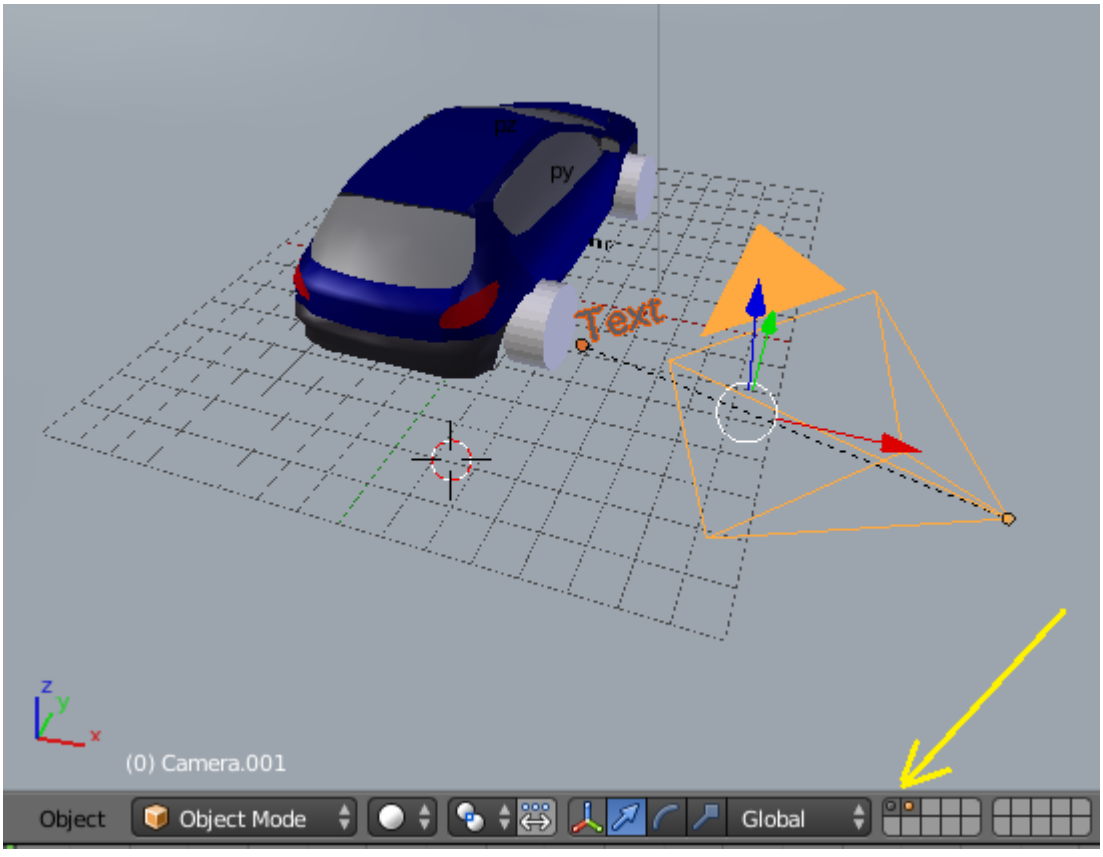
А вот настройки материала:

Одна особенность готовых текстурированных моделей 3ds может вас неприятно удивить. Если вы что-то пропустили в настройках материала, то при создании EXE файла и переносе его на другой компьютер вы можете обнаружить, что некоторые текстуры вовсе не отображаются. Поскольку текстуры лучше накладывать на прозрачный материал, напоминаю вам его настройки:

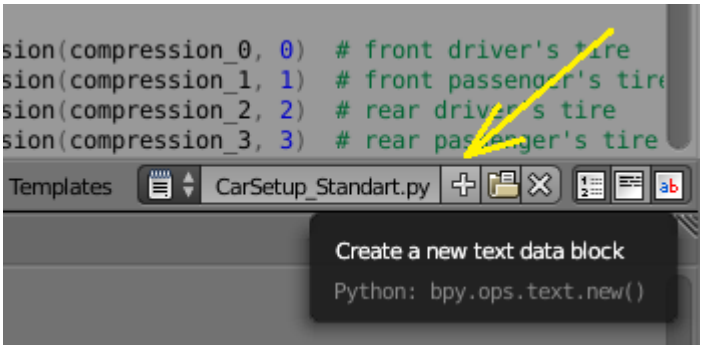


Но лучше всего вообще не использовать текстуры в модели (за исключением, может, колёс). Раскрасьте сами и вы будете уверены, что модель всегда и везде будет выглядеть одинаково. Я так и поступил. Оставил текстуру у колёс, а остальное раскрасил:

Теперь займёмся цифровым индикатором скорости. Он нам необходим для наглядности в наших дальнейших экспериментах. Для этого переходим на второй слой и создаём камеру. Под ней создаём текстовый объект и прикрепляем его к камере по принципу «потомок-родитель». Теперь в первом слое удаляем камеру. Включаем два слоя одновременно и поворачиваем камеру на автомобиль, расположив её на нужном расстоянии. Не забудем в логике прикрепить к автомобилю новую камеру, как мы делали с предыдущей. Подготовительные работы закончились:

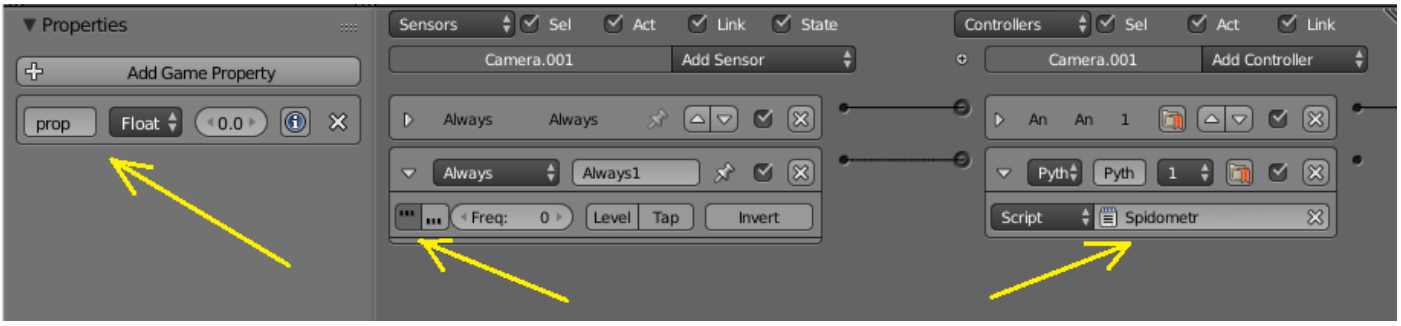


Давайте создавать логику текстового объекта. Выделим только камеру и перейдём в Game Logic. Далее, справа жмём плюсик и пишем скрипт Spidometr:




```python
import bge
cont=bge.logic.getCurrentController()
obj = cont.owner

scene = bge.logic.getCurrentScene()# получить текущую сцену
objList = scene.objects            # получить список объектов
car = objList["Car"]               # получаем об машины
text = objList["Text"]             # получаем текстовый об

vec=car.localLinearVelocity        # получить локальный вектор скорости
obj["prop"] = vec[1]/30       # присваиваем переменную Float и делим на 30

speed = obj["prop"]              # присваиваем обычную переменную

if speed<-0.001:       # если скорость меньше 0.001 то присваиваем 0
    speed = 0

# переводим скорость в понятные цифры
speed = speed*100
speed = int(speed)

# конвертируем цифры в строку
text.text = str(speed)         # выводим на экран
```

И добавим игровое свойство:



Если мы включим вид с камеры и поедем, то текстовый объект будет показывать скорость. Правда, только при движении вперёд ☺ Но этого достаточно для наших экспериментов.

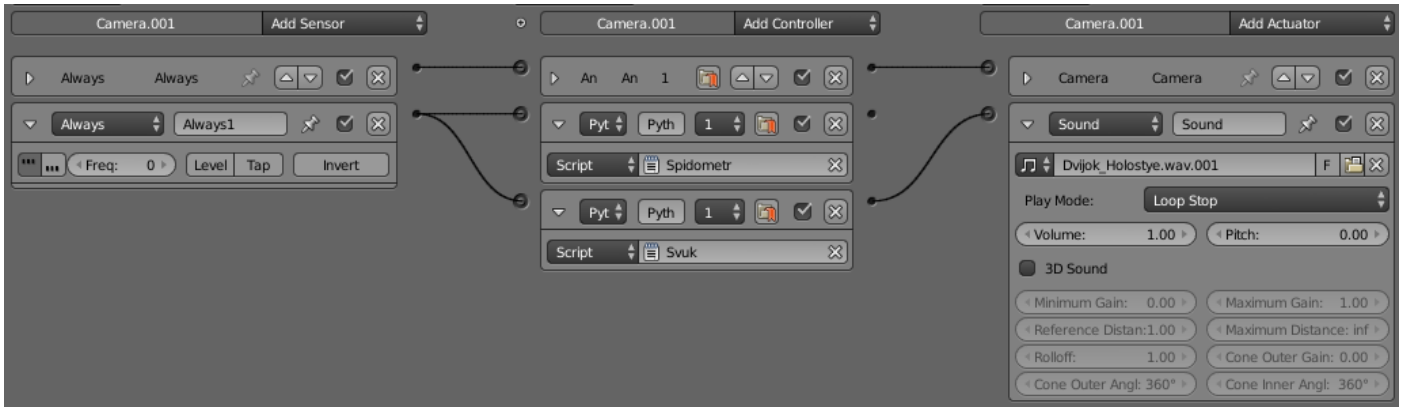
Так для чего же нам спидометр? Для красоты? Нет. Например, он необходим для отслеживания высоты тона звука двигателя. А ещё, для ограничения скорости движения. Если не ограничивать скорость, то она может расти до бесконечности, при движении по прямой. В реальности такого просто не может быть ☺.

Снова выделяем камеру и переходим в логику. Создаём новый скрипт Svuk:

```
import bge
cont=bge.logic.getCurrentController()
obj = cont.owner
act = cont.actuators["Sound"]

scene = bge.logic.getCurrentScene()# получить текущую сцену
objList = scene.objects            # получить список объектов
car = objList["Car"]               # получить объект "Car"

vec=car.localLinearVelocity        # получить локальный вектор скорости
obj["prop"] = (vec[1]/10)       # присваиваем переменную Float и делим на 10

speed = obj["prop"]             # присваиваем обычную переменную

if speed > 2:           # если скорость больше 2 то
    speed = 2           # скорость равна 2
if speed < 0.5:         # если скорость меньше 0.5 то
    speed = 0.5         # скорость равна 0.5

act.pitch=speed         # изменить свойство Pitch звукового актуатора
cont.activate(act)      # активировать актуатор
```

View Text Edit Format Templates Svuk

Добавим ещё один контролёр Python и актуатор Sound:



Как вы должны были заметить, файл звука двигателя в формате WAV. Из урока с подводной лодкой вы уже знаете, почему. Этот файл тоже в папке Car_Blend. Осталось лишь ограничить скорость движения. Ну… Это совсем просто ☺ В скрипте Spidometr мы уже применяли функцию localLinearVelocity, которая может и получать и устанавливать локальную линейную скорость. Соответственно в этом же скрипте мы в самом конце лишь допишем две строки кода:

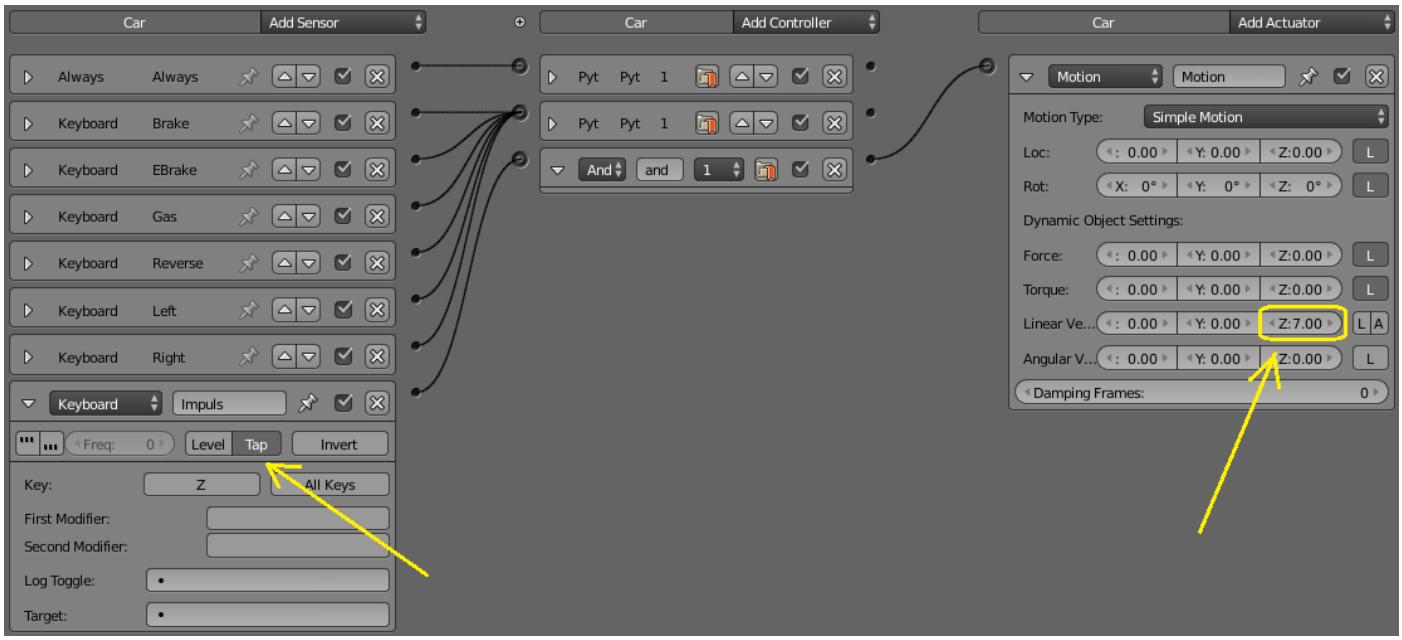
```
if speed > 65:  # если скорость больше 65 то

    car.localLinearVelocity = [ 0.0, 19.5, 0.0]

    # линейная скорость по оси Y равна 19.5
```

```
import bge
cont=bge.logic.getCurrentController()
obj = cont.owner

scene = bge.logic.getCurrentScene()# получить текущую сцену
objList = scene.objects            # получить список объектов
car = objList["Car"]               # получаем об машины
text = objList["Text"]             # получаем текстовый об

vec=car.localLinearVelocity        # получить локальный вектор скорости
obj["prop"] = vec[1]/30       # присваиваем переменную Float и делим на 30

speed = obj["prop"]             # присваиваем обычную переменную

if speed<-0.001:       # если скорость меньше 0.001 то присваиваем 0
    speed = 0

# переводим скорость в понятные цифры
speed = speed*100
speed = int(speed)

# конвертируем цифры в строку
text.text = str(speed)        # выводим на экран

if speed > 65:  # если скорость больше 65 то
    car.localLinearVelocity = [ 0.0, 19.5, 0.0]
    # линейная скорость по оси Y равна 19.5

View Text Edit Format Templates Spidometr
```

Как мы получили 19.5 ? Если для спидометра мы вектор делили на 30 и умножали на 100, то в ограничителе нужно сделать наоборот: разделить на сто и умножить на 30. А 65 я

нашел сугубо опытным путём. Я «прокатился» и заметил, на каком числе звук перестаёт изменяться. Это было 65. Значит это максимальное возможное число. Вот и всё ☺.

Что ещё хотелось бы сделать? Иногда машина может перевернуться и остаться на крыше. Как выйти из этого положения? Конечно, можно начать заново игру ☺ Но можно поступить практичнее. Нужно придать ей короткий импульс вертикально вверх в мировых координатах. Тогда, возможно, она станет на колёса. Сделаем это по нажатии на клавишу Z. Выделим Car и в логике добавим сенсор для клавиши Z, контролёр и актуатор движения:



На этом мы завершим этот урок. Основной принцип вам уже должен быть понятен. Далее можно будет говорить лишь о логике игры: добавлении соперников или триггеров прохождения определённых участков пути… Но это уже другая тема ☺.

Составил **Niburiec** для сайта **http://blender-game.ucoz.ru**