ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

ΣΧΟΛΗ ΕΠΙΣΤΗΜΩΝ & ΤΕΧΝΟΛΟΓΙΑΣ ΤΗΣ ΠΛΗΡΟΦΟΡΙΑΣ SCHOOL OF INFORMATION SCIENCES & TECHNOLOGY

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ DEPARTMENT OF INFORMATICS

Computer Graphics Final Project

Winter Semester 2017-2018

Shooting Cans!

Write a program in C++ using OpenGL[1], based on the OpenGL lab course code, which implements a first-person shooter. As a baseline, the goal of the game is to eliminate as many "cans" (robots) as possible before being hit by them. You can implement any other game logic or augment this simple setup with more advanced features and include power-ups, a user respawn system, bosses, complex AI, or one-on-one functionality (split screen or networked).

Screenshots from a mockup implementation of the game.

[1] Alternatively, you can program the application in Android or another platform/programming language. Contact the instructor for eligible alternatives.

The cans are randomly spawned from any valid (unoccupied) location chosen from a list of predetermined locations. They represent the enemy models and the player must hit them with a bullet before they manage to either shoot back (optional) or reach the player. When a can is spawned, it has a random orientation around its Y axis. The can immediately moves forward along its facing direction. This forward direction is gradually corrected towards a “goal” direction that faces the player. With a probability that decreases as the can approaches the player, the “goal” may randomly change.

In every frame, the player and each can should check their position using collision detection (see respective section below). If a collision prevents them from moving to the estimated new position, they are immobilized for the current frame. Collisions must be checked between: a) the player and the environment, b) the player and each can, c) among cans.

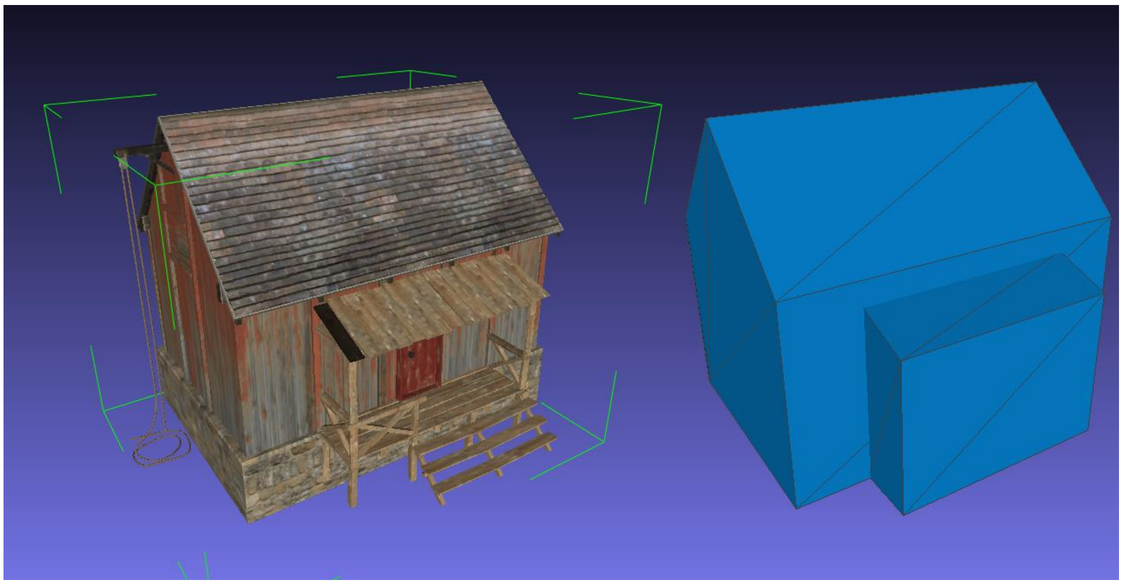
Player shooting takes place by firing the handgun the user is equipped with (see images above). Once fired, by triggering the event with a mouse button, the ray connecting the nozzle towards the front vector of the gun is checked for a hit with the collision hull of each can. If a hit is found, the corresponding can is removed from play. New cans are periodically spawned, regardless of the kill rate. When a can reaches a predetermined distance from the player the game is over, so the player must constantly move to avoid the cans and reduce their numbers.

The Game Level

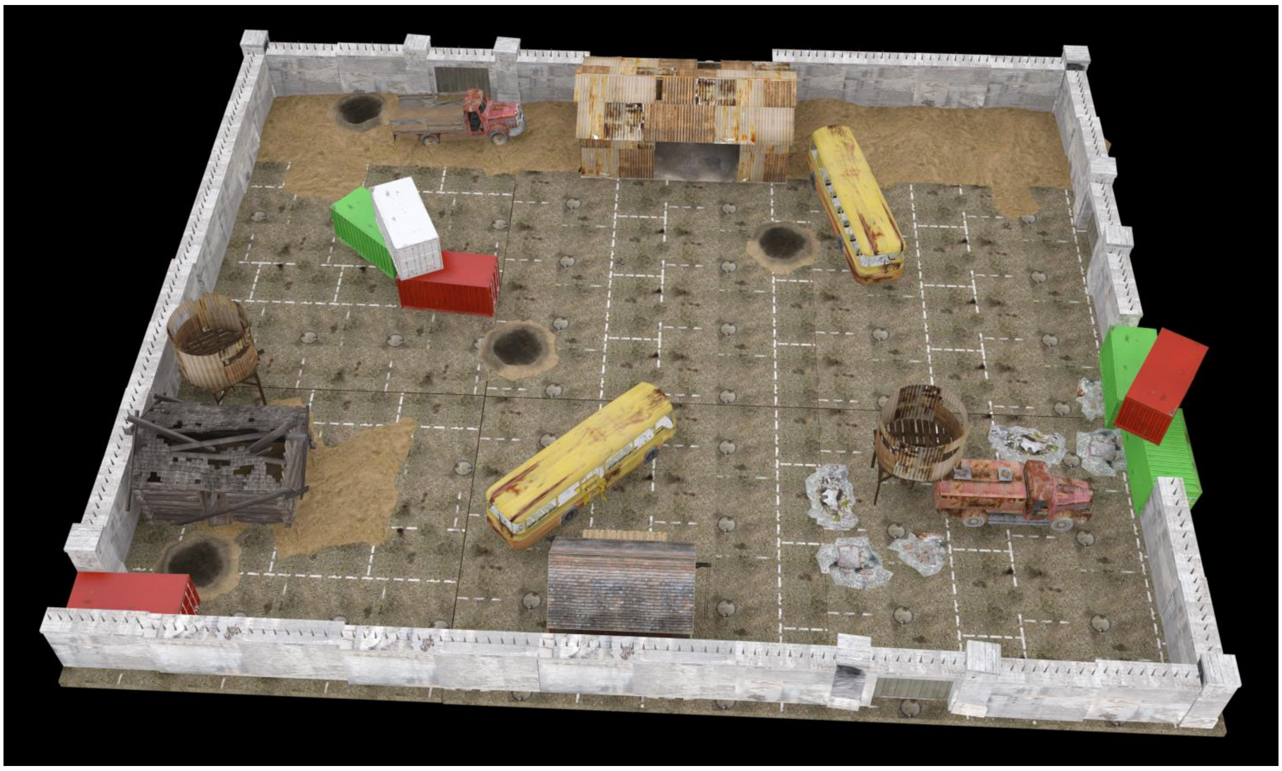
The static environment, where all the action takes place, should be composed from a list of assets provided in the data files. The 3D models represent scenery parts, obstacles and ground modelling. The 3D models are in the Wavefront OBJ format and come with a much simplified version of themselves, named as [ASSET NAME]-Collision.obj. This secondary object is the collision hull for the detailed (renderable) version of the geometry and should participate in ray-object intersection queries. The collision hulls are not intended for rendering, but one can visualize them along with the normal geometry for debugging purposes. Make sure to apply the identical object-to-world transformation to them (in the host code, e.g. using the glm matrix – vector multiplication functionality) as the one applied to their rendered counterparts (in the GPU) so that the collisions detected are in line with the visual result.

The scene where all the action takes place should be composed from a list of assets provided in the data files. The 3D models represent scenery parts, obstacles and ground modelling. The 3D models are in the Wavefront OBJ format and come with a much simplified version of themselves, named as [ASSET NAME]-Collision.obj. This secondary geometric representation should be used with the intersect_triangles() and intersect_triangles_collision_only() functions of the provided isect.h header file to report ray-object intersection tests.

All models provided (including the can parts and the handgun of the player) are measured in **meters.**

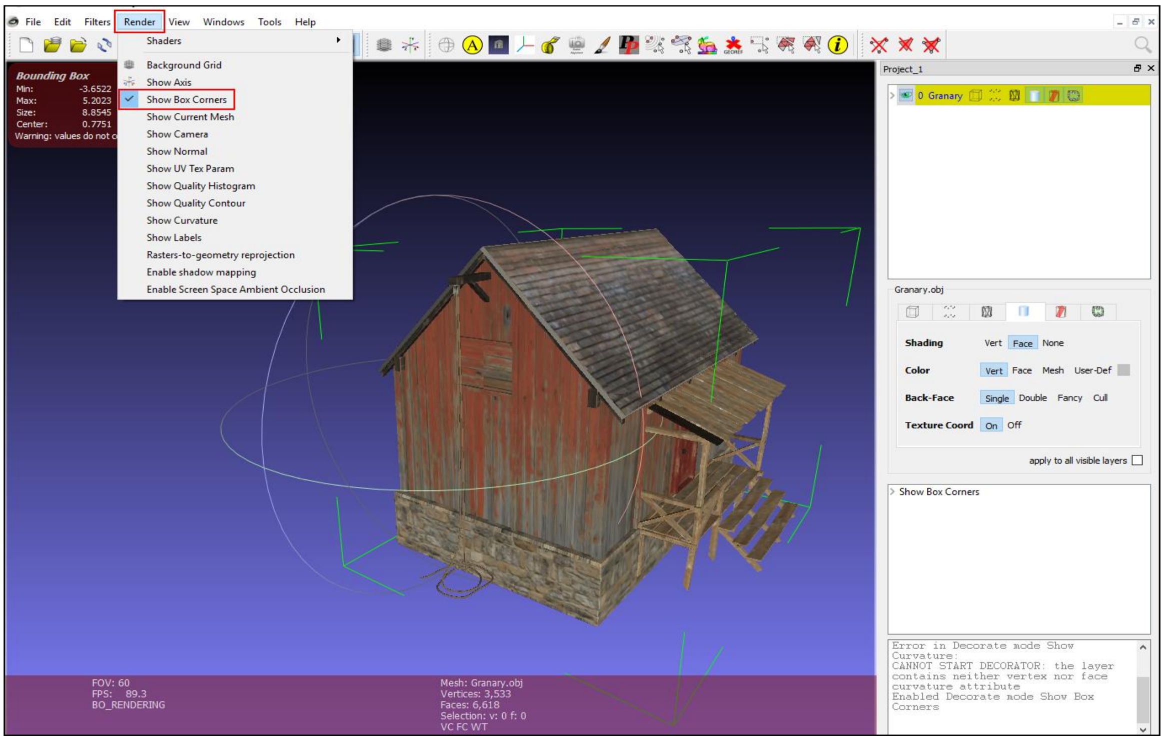


Collision hull of the model Granary.obj (Grannary-Collision.obj)



An example of a game level designed with the provided assets.

Since the assets comprising the environment are numerous, a detailed listing and individual dimensions are not provided here, for brevity. However you can download the free software MeshLab (http://www.meshlab.net/) to load and preview the models, as well as query them for basic dimensioning properties as shown below:



```
Bounding Box
Min:      -3.6522
Max:       5.2023
Size:      8.8545
Center:    0.7751
Warning: values do not c
```

```
FOV: 60
FPS:   89.3
BO_RENDERING
```

```
Mesh: Granary.obj
Vertices: 3,533
Faces: 6,618
Selection: v: 0 f: 0
VC FC WT
```

```
Error in Decorate mode Show
Curvature:
CANNOT START DECORATOR: the layer
contains neither vertex nor face
curvature attribute
Enabled Decorate mode Show Box
Corners
```

Using MeshLab to preview objects and read their dimensions and position.

Place each model one or more times in the environment by loading it and transforming it according to the example code in the Lab projects. All static scenery should be placed in a flat hierarchy (i.e. all objects are first-level ones, without dependent transformations on other nodes). If you wish, you can automate the process by implementing a custom scene description file format similar to the one shown below, where you would be able to declare which object to load and what transformation to apply to it outside the source code, so that you won't have to recompile your application every time you tweak the position of an object.

```
Barn 1.0 1.0 1.0 3.14159 0.0 1.0 0.0 10.5 0.0 -2.4

WallSegment1 1.0 1.0 1.0 0.0 0.0 1.0 0.0 1.5 0.0 12.6

WallSegment1 1.0 1.0 1.0 0.84147 0.0 1.0 0.0 1.5 0.0 12.6
```

Object name — Scale coefs. — Rotation angle and axis — Translation

The .obj extension
is added automatically

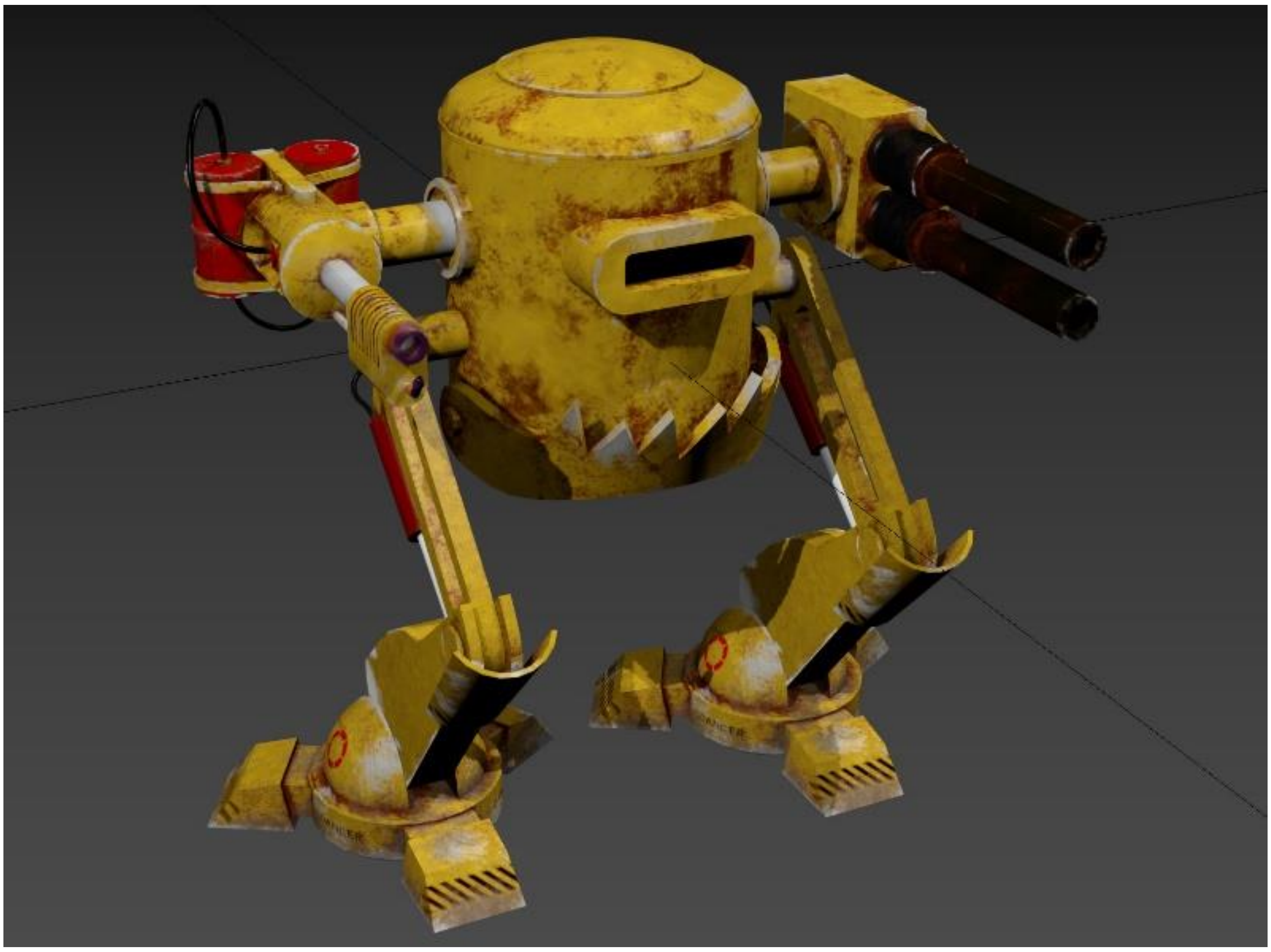
The _collision version is also loaded if available and transformed using the same matrix.

Example scene description file for easier loading of the environment's geometry

For easier composition of the environment, you may utilize any of the freely available modeling software such as 3D Studio Max, Maya [2] or Blender to either position the assets in the scene and then export the finalized environment as 1-2 composite OBJ models or simply record the transformation required to bring the models in the desired pose. If you choose to do this, you MUST also load and transform the collision hulls to properly correspond with the edited environment.

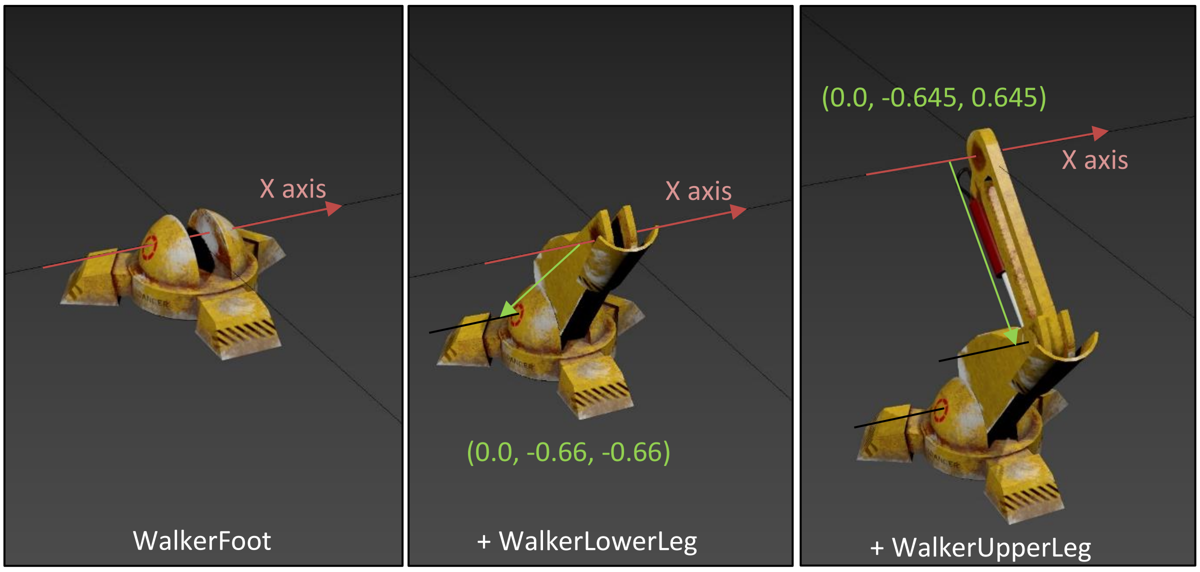
The Can

The enemy model is a bipedal mech (called a "can", due to its similarity to a food can or a canister), equipped with two weapons, one flame thrower and one dual cannon. The can is an articulated model so that you can independently animate its parts while it moves. The instructions below show how to assemble it from its constituent parts so that you can also animate its part relative to its parent.

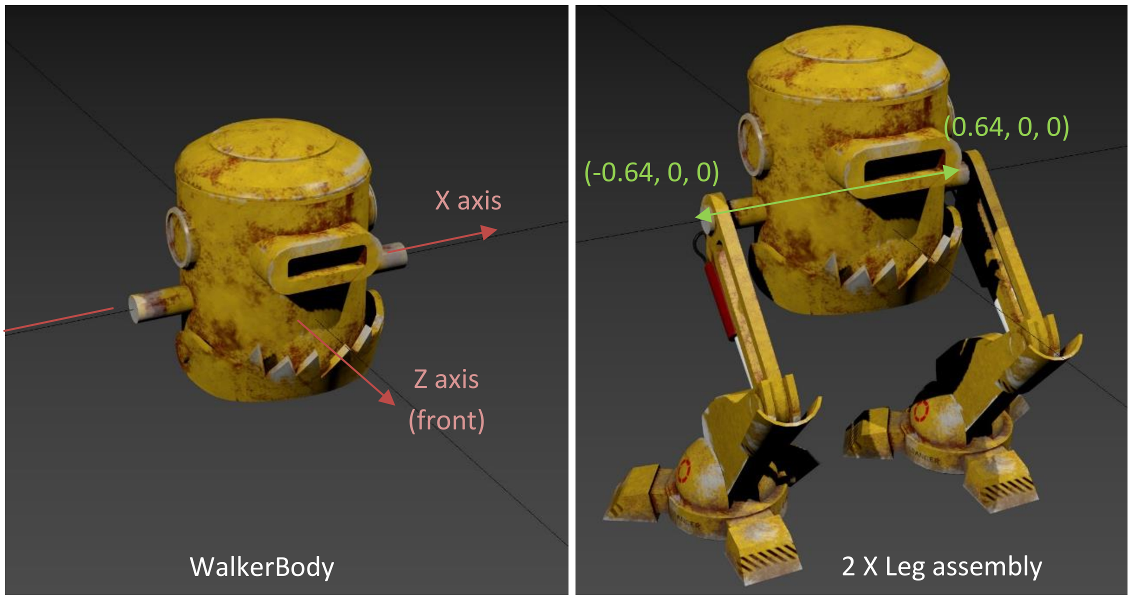


The assembled can model

[2] Students can download a free copy from Autodesk



Mech leg assembly. Each part uses the local X axis for defining its rotation relative to the parent part. Green arrows indicate the relative offset.

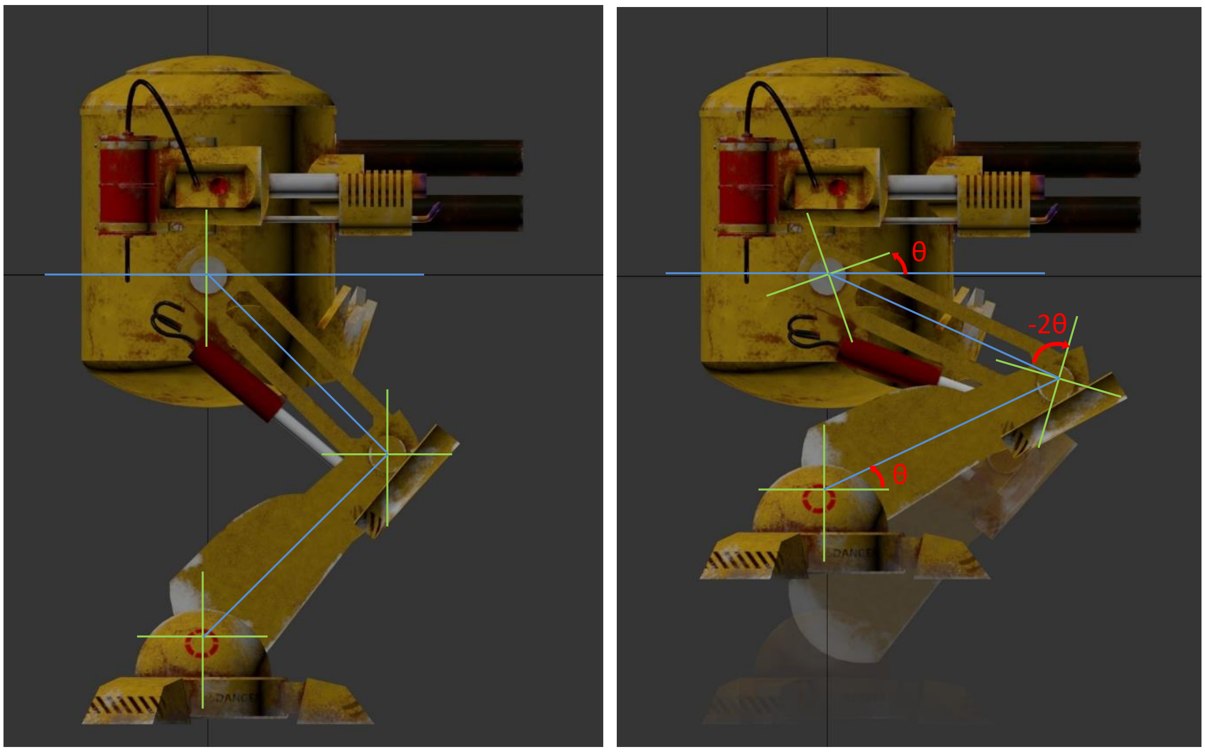


Lower mech assembly. The two identical leg sub-assemblies are already aligned with their pivot axle so the only thing required is to translate them on the left and right side of the mech.

The reference object for the articulated object hierarchy of the can is the WalkerBody part. The body is centered at its local reference frame so that the X axis passes through the torso axle that is the mounting point for the legs.

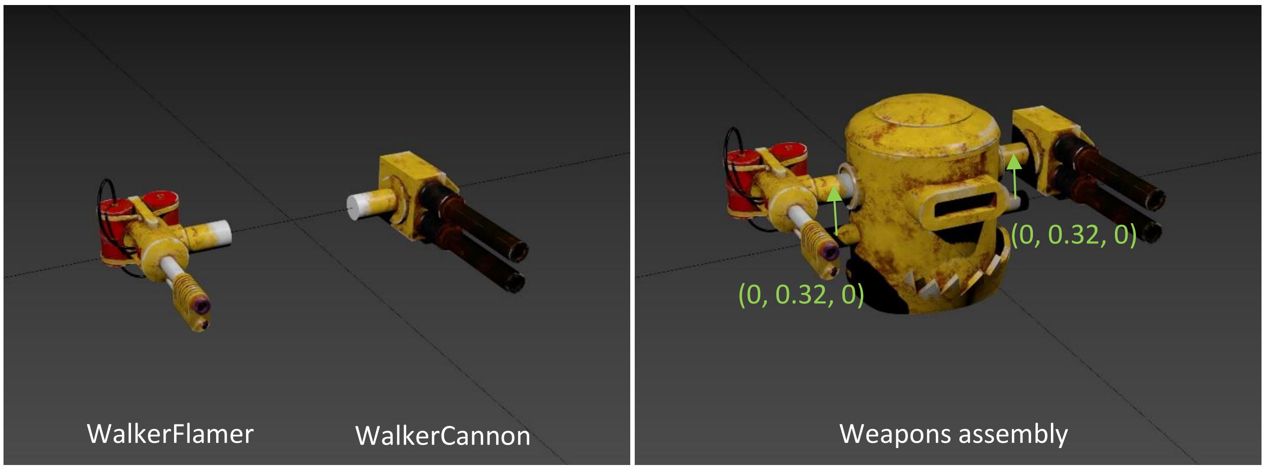
The two legs are identical and only differ by an offset in the X axis (see Figure above). Each leg consists of 3 parts, the foot, the lower leg and upper leg. The foot joint (ankle) can be rotated around the X axis and then translated to take its position relative to the knee joint of the lower leg (see top figure, left and middle). In the same manner both the lower leg and the foots can be rotated around the knee axle and placed relative to the torso joint that allows the full leg to be also rotated around its X axis (see top figure – right).

Mimicking a walk cycle while the mech moves is easy if one is to take advantage of the articulated leg structure; Complex motion can be simulated by rotating one of the joints around its local X axis by an angle $\theta = f(v,t)$, where v is the current linear velocity of the mech and t is the current application time. The rotation of the other joints of the leg are dependent on this angle to cohesively move the leg parts, as shown in the example of the following figure. An example of a function for angle θ could be [3] : $f(v,t) = \frac{\pi}{12} + \frac{\pi}{12}\sin(2\pi t v)$.



Walk-stomp motion example.

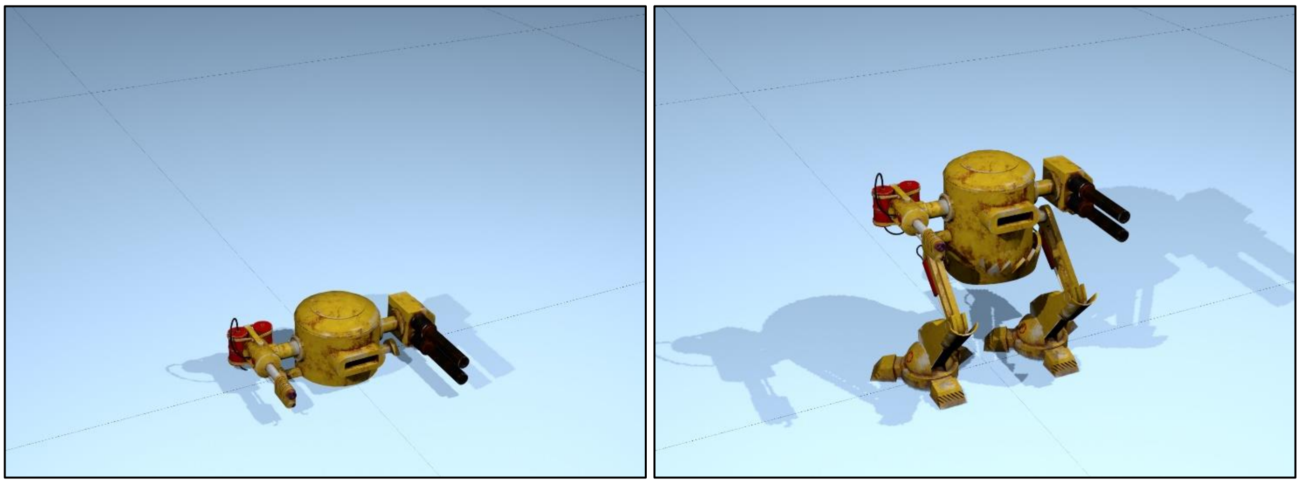
The weapons are two separate objects that can be rotated around their local X axis prior to being offset to their respective sockets as shown in the example below:



Weapons assembly. The original position of the flamer and cannon (left) enable the rotation of the weapons around their mounting axle (local X axis). To finalize their position relative to the can body, a small offset is required in the Y axis (right).

[3] We assume here that the mechs start moving at spawn time and do not cease to do so until they are removed from play.

Finally, the overall bearing of the mech when loaded (i.e. its "front" direction) is along the Z axis. A suggested function for changing its bearing is discussed below. The mech has to be translated as a whole to stand on the ground (feet at y=0) by (0.0, 1.63, 0.0):



Bringing the mech to a standing level.

The can normally moves forward along its front $\mathbf{d}_{front}$ direction, unless the new position brings it too close to an obstacle. See the Detecting Collisions section for details about the later. If a collision is detected, the position of the mech is not advanced, otherwise the new position $\mathbf{p}'$ is easily calculated according to the equation:

$$\mathbf{p}' = \mathbf{p} + \Delta t \cdot v \cdot \mathbf{d}_{front} \qquad (1)$$

where $\mathbf{p}$ is the previous position, v the linear velocity of the mech and Δt is the time elapsed between the current calculation and the previous one. Normally, this time interval corresponds to the time measured between the current and previous frames, assuming an update of the game's state occurs in every frame.

The forward direction $\mathbf{d}_{front}$ is also updated in each frame based on two rules: a) the mech must chase the player, so it has a "goal" direction $\mathbf{d}_{goal}$ pointing towards the player and b) the mech randomly decides to "walk off" towards some other target with a probability P_{roam}. This probability can be dependent on the distance of the player to the current position of a mech, so that distant mechs seldom try to follow the user, while close ones quickly converge to him or her. To avoid a jerky movement, the front direction is gradually shifted towards the current goal direction and also the goal direction is not updated in every frame, allowing a lag in the mech's response and a smoother change of direction.

Given the current goal direction $\mathbf{d}_{goal}$ of the mech, the new (normalized) forward vector $\mathbf{d}'_{front}$ is estimated as:

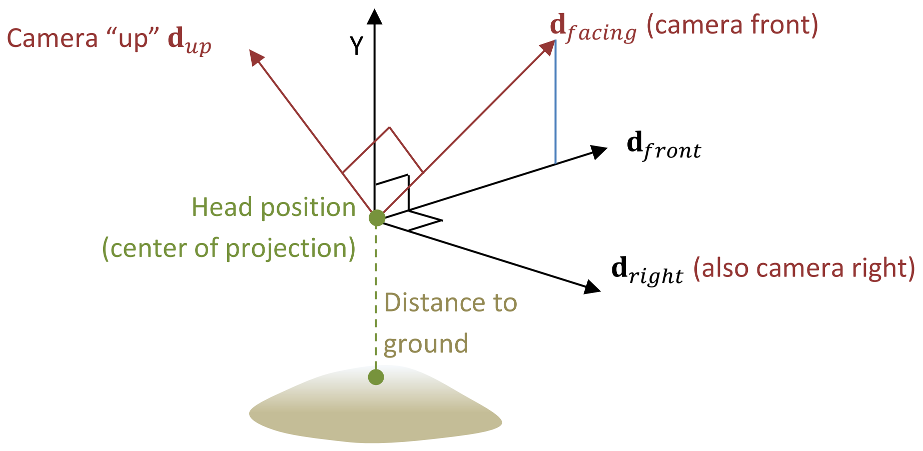
$$\mathbf{d}'_{front} = normalize\big((1-\gamma)\mathbf{d}_{front} + \gamma\mathbf{d}_{goal}\big) \qquad (2)$$

where γ is the *responsiveness* factor with values strictly in the range $0 < \gamma < 1$ (notice that γ should never be equal to 0 or 1).

Both the $\mathbf{d}_{goal}$ and $\mathbf{d}_{front}$ directions **should lie on the XZ plane**. All moving objects navigate the scene horizontally and only adjust their vertical position according to their collision detection with the ground (see Detecting Collisions next).

Navigating the Environment

The game is a first-person one and therefore the viewpoint is centered at the player. The player can freely look around his or her position, by defining a new facing direction $\mathbf{d}_{facing}$ using the mouse but **always moves on the XZ plane relative to the same forward bearing as the facing vector**. Forward or backward movement along the front direction $\mathbf{d}_{front}$ is done with the cursor keys, which also accommodate the "strafe" action, i.e. the sideways motion along the "right" vector $\mathbf{d}_{right}$, which is always perpendicular to the player's front vector. Shooting uses the facing vector and the projectile is assumed to leave from a weapon lower and to the right of our viewpoint. The player's origin is assumed to be at the head position $\mathbf{p}_{player}$ that coincides with the center of projection (viewpoint). The player's origin is adjusted in every frame to be at a fixed distance from the ground using collision detection (see respective section next).


Camera "up" $\mathbf{d}_{up}$
Y
$\mathbf{d}_{facing}$ (camera front)
$\mathbf{d}_{front}$
Head position
(center of projection)
$\mathbf{d}_{right}$ (also camera right)
Distance to
ground

The player setup.

Let f, s be two scalars representing the (f)orward and (s)trafe movement of the player with values $-a, 0, a$. $-a$ corresponds to backward and left movement, 0 to no movement and a to forward and right movement. The magnitude of the motion offset a is a user-defined parameter and represents the linear velocity of the player. Let also Δx and Δy be the screen offsets of the mouse as it moved between the previous and current frame and b the rotation speed of the player. The vectors used in the motion and viewing transformations are updated as follows:

$$\mathbf{d}'_{facing} = normalize(\mathbf{d}_{facing} + b\Delta x\mathbf{d}_{right} + b\Delta y\mathbf{d}_{up}) \quad (3)$$

$$\mathbf{d}'_{right} = normalize(\mathbf{d}_{facing} \times Y) \quad (4)$$

$$\mathbf{d}'_{up} = \mathbf{d}'_{right} \times \mathbf{d}'_{facing} \quad (5)$$

For the motion calculations we now have the updated $\mathbf{d}_{facing}$ and $\mathbf{d}_{right}$ and given the Δt time interval between updates, the new forward motion vector and position is determined by:

$$\mathbf{d}_{forward} = normalize(\mathbf{d}_{facing}.x, 0, \mathbf{d}_{facing}.z) \quad (6)$$

$$\mathbf{p}'_{player} = \mathbf{p}_{player} + f\Delta t\mathbf{d}_{forward} + s\Delta t\mathbf{d}_{right} \quad (7)$$

The position update in Eq. 7 only occurs if no collision is detected.

Detecting Collisions

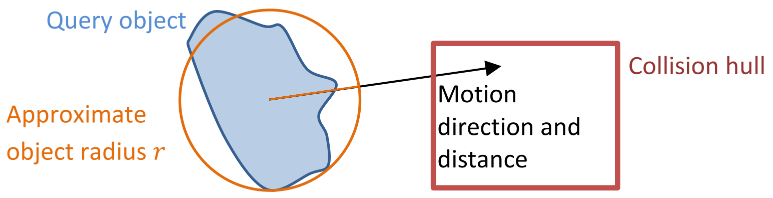
For the detection of collisions and the measurement of a distance to a target geometric entity, a header-only library is provided (isect.h), which implements two functions, one for determining whether the target is within a given distance (faster function with early exit) and one for measuring the exact distance to the closest target geometry surface. The following code snippets apply these two functions on geometry loaded according to the lab example code to determine closeness to the collision hull of a mesh and also determine the exact distance to it. All collision detection operations that we are going to use in this project assume a test position and a direction. Therefore, we test a ray (origin and direction) for intersection with the target geometry. In the proximity test, we check whether there exists an intersection of the ray and the geometry that is closer than a user-defined radius to the ray origin. When we want the exact distance to the target, we try to find the closest ray hit to the geometry, so the query is exhaustive.

To set up our geometry for collision, we must first load the mesh that corresponds to the collision hull and copy vertices of its triangles into a separate linear buffer (the case in the particular code example) or use the provided storage (mesh->vertices) directly. Copying the vertices in a new vector is useful when we have many static objects (e.g. the static environment of a game level). These can be transformed to their final position using the corresponding meshes' transformation matrix, compacted into a single vector and get tested altogether without transforming them each time, as is the case in the collision tests here.

```
OBJLoader loader;
auto mesh = loader.load("Assets/WalkerBody-Collision.obj ");
std::vector<glm::vec3> collision_hull;

if (mesh != nullptr)
{
    GeometryNode * collision_geometry = new GeometryNode();
    collision_geometry->Init(mesh);
    for (auto item : mesh->vertices)
        collision_hull.push_back(item);
}
```

When determining whether a moving object is about to get too close to another geometric entity, we must test the current position of the object together with an offset vector to the new (intended) position. So the resulting query ray has a position equal to the moving object center, a direction vector equal to the normalized intended move direction offset and a test radius equal to the sum of the moving object's "radius" plus the distance it intends to cover.



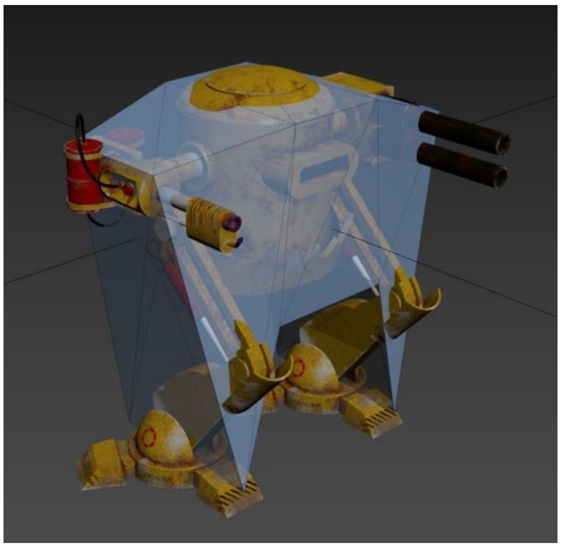
Checking for collision between a moving object (blue) and an obstacle (red) using the `intersect_triangles_collision_only` *function. The test radius is the sum of the object's approximate radius and the motion distance.*

The following function example tests for collision a moving entity (object, camera etc.) centered at pos with a normalized direction dir and a query radius corresponding to the sum of the object's approximate radius plus the motion distance, respecting potential geometric transformations M of the collision hull.

```
bool collided(float radius, const glm::mat4 & M,
              const std::vector<glm::vec3> & collision_hull,
              const glm::vec3 & pos, const glm::vec3 & dir)
{
    std::vector<glm::vec3> transformed_hull(collision_hull);
    std::transform( transformed_hull.begin(),
                    transformed_hull.end(),
                    transformed_hull.begin(),
                    [&M](glm::vec3 v) {return glm::vec3(M*glm::vec4(v,1.0f)); }
                  );
    return intersect_triangles_collision_only(pos, dir, transformed_hull.data(),
           (unsigned int) transformed_hull.size(), radius);
}
```

For determining the exact distance to the collision hull, the following code can be used. This functionality is required for a) adjusting the position of the user and cans to match the ground elevation, b) determine the hit point of a projectile in order to spawn an explosion object or other effect there.

```
float distance_to_target(const glm::mat4 & M,
                        const std::vector<glm::vec3> & collision_hull,
                        const glm::vec3 & pos, const glm::vec3 & dir)
{
    std::vector<glm::vec3> transformed_hull(collision_hull);
    std::transform( transformed_hull.begin(),
                    transformed_hull.end(),
                    transformed_hull.begin(),
                    [&M](glm::vec3 v) {return glm::vec3(M*glm::vec4(v,1.0f)); }
                  );
    glm::vec3 hit;
    glm::vec3 n_hit;
    bool res = intersect_triangles(pos, dir, transformed_hull.data(),
           (unsigned int) transformed_hull.size(), hit, n_hit);
    if (!res)
        return -1.0f; // a negative value signifies a miss
    return glm::distance(pos, hit);
}
```



The collision hull of the mech is bound to its body part (WalkerBody-Collision.obj) and is not animated along with the leg or weapon movement but rather approximates their possible movement extent.

To vertically adjust the position of any moving entity relative to the ground geometry elevation, one can shoot a ray from the center of the object of the moving object towards -Y and measure the distance to the hit geometry using the function shown above. If no hit is detected (perhaps due to a gap in the ground), the height of the moving object is not adjusted. Otherwise, the Y coordinate of the moving object is adjusted according to the difference between the desired distance to ground and the measured one.

Lighting

The scene should appear dark and the only prevalent light source is supposed to be a flashlight carried by the player. The static lighting should therefore be either a directional or distant point overhead light source shooting downwards a faint bluish (nightly) glow or a distant spot light with a shadow map with heavily blurred shadow edges (wide PCF kernel).

The primary light source, i.e. the flashlight, should be modelled as a spotlight with a circular emission hotspot, distance attenuation and a shadow map attached. The light position should follow the player and to properly match a realistic scenario, it should be located to the lower left side of the player center of projection. Keeping the flashlight positioned about 0.4m or more to the left of the player also gives enough disparity between the camera frustum and the shadow map frustum to have some nice visible shadows that help delineate the objects against the background. The flashlight spotlight direction should point parallel to the $\mathbf{d}_{facing}$ vector.

Finally, the interested student can add a distance darkening effect to the entire environment (like black fog effect) to further obscure the distant geometry and enhance the eeriness of the environment and disorientation of the player.

Enhancements

Feel free to add various enhancements, either visual, inspired by the Lab examples, or otherwise, including night vision mode, crosshair, power-ups, visible projectiles, laser beam guides (you can use the `intersect_triangles` function to build a line between the gun tip and the hit point), enhanced AI, better game level assets or sound effects and music (OpenAL is a good start for this).

Happy Coding!