denden game engine

B. Burak Payzun

Last Updated: October 6, 2021

This document describes the capabilities of the `denden` and its features. Keep in mind that `denden` is not even close to a fully fledged game engine and should only be used for research purposes. This guide will help the reader to quickly start with the help of examples shipped in `denden`.

Contents

1 Introduction

1.1 How does denden work?

denden currently works on a single thread which means codes run one after the other so please don't shoot yourself in your foot by doing things like infinite loops (or even time consuming loops).

1.2 App

App creates the renderer window, the input manager and the scene for you. What you need to do is derive a class from App and override scene_setup() method and call the App::run() function from main().

1.3 Scene

Scene is the part of the game engine that combine all the other parts of the project like physics engine, and objects. By overriding App::scene_setup() in your project you set up the scene to your needs.

1.4 Game Objects

In denden **GameObject**s are the things that represent almost anything in the game. Game objects have components (which is discussed in 2.1) that you use to define your game objects.

Important member functions in game objects are `add_component<T>()` and `get_component<T>()`

Example usage:

```
std::shared_ptr<GameObject> obj = scene->create_gameobject("obj");
// add_component
obj->add_component<Collider>(new CapsuleCollider(0.5, 2));
// get_component
std::share_ptr<GameObject> ref = obj->get_component<Collider>();
```

1.5 Physics

The Physics class uses wrapped bullet3 as the physics engine. Objects with RigidBody components are automatically added to the rigidbody list and affected by the physics. Bullet3 by itself is very robust, but isn't fully wrapped in denden as of now (yet it does the job).

1.6 Drawing GUI elements

For now, denden uses imgui to draw gui elements on top of the window, but none of it is wrapped and changing the layout isn't supported modularly

in denden right now. But if you really want to change it, you can change it in `src/app.cpp App::run()`.

2 Components

2.1 What are components and how to use them?

Every **game object** consists of **components** that define the behaviour of the object. `denden` comes with some essential components in it, but if you want (which you will most likely need), you can also create your own component. You can refer to section 2.3 for that.

Components are used by calling **add_component<T>()** and **get_component<T>()** methods on game objects.

An example to this is given below.

```
// Creating the game object
    std::shared_ptr<GameObject> player = scene->create_gameobject("
    player");

// Adding a RigidBody component with a mass of 0.5
    /* std::shared_ptr<RigiBody> rb = */ player->add_component<
    RigidBody>(new RigidBody(0.5));

// Getting the RigidBody component on the player game object.
    std::shared_ptr<RigiBody> rb = player->get_component<RigidBody
    >();

// Using component methods.
    rb->freeze_rotation_x(true);
```

add_component<T>() return the component it create, but that part is commented out to show get_component<T>() too.

2.2 Essential Components

Below is the list of components that come with `denden`.

2.2.1 Transform

The **Transform** component holds the position, orientation and scale of the game object. Every game object is given a Transform component by the **Scene** for convenience.

2.2.2 Model

The **Model** holds the mesh data, to be used in rendering. Models with .obj file type can be loaded with the `Model(std::string path)` constructor.

2.2.3 RigidBody

Game objects with **RigidBody** are affected by the physics engine. RigidBody currently needs a **Collider** to work.
Can be constructed by `RigidBody(double mass)`.

2.2.4 Collider

Collider defines the volumetric properties of bodies in physics engine. **Collider** currently has 6 subclasses.

- `BoxCollider(float x, float y, float z)`
- `SphereCollider(float radius)`
- `CapsuleCollider(float radius, float height)`
- `CylinderCollider(float x, float y, float z)`
- `ConeCollider(float radius, float height)`
- `MeshCollider(Model model)`
  Please note that using the MeshCollider is costly as Meshes are automatically generated from model vertices. It is not recommended to use them frequently.

2.2.5 Camera

In denden, the **Camera** is also a component. denden can only render a single screen from a camera at a time, the active camera should be set in scene_setup() as:

```
1 void scene_setup() override{
2     // Creating the camera
3     std::shared_ptr<GameObject> camera_object = scene->
      create_gameobject("Main Camera");
4     camera_object->add_component<Camera>(new Camera(glm::vec3(0.0f,
       0.0f, 15.0f), glm::vec3(0.0f, 0.0f, -1.0f), glm::vec3(0.0f,
      1.0f, 0.0f), glm::radians(75.0f), (float)renderer->window_width
       / (float)renderer->window_height, 0.1f, 120.0f));
5     //..
6     scene->set_active_camera(camera_object->get_component<Camera>()
      );
```

Can be constructed by

```
1 Camera(glm::vec3 pos, glm::vec3 target, glm::vec3 up, float
      fov_angle, float aspect_ratio, float near, float far)
```

2.2.6 LamdaScript

LambdaScript is a flexible component that runs 2 lambda functions, start and update.

Start

Is given to LambdaScript in the constructor, runs only once when the component is added to the game object. Constructor is as follows:

```
LambdaScript(std::function<void(LambdaScript*)> func = [](
    LambdaScript* ref)
```

Update

This function is called every frame before the rendering and physics step. The function gets a LambdaScript* as a reference to self and a float that is the delta time since the last frame. Can be set as follows:

```
// Creating the LambdaScript component
std::shared_ptr<LambdaScript> player_lambda = player->
    add_component<LambdaScript>(new LambdaScript());
// Setting up the update function using set_update_func(std::
    function<void(LambdaScript*, float)> func)

player_lambda->set_update_func(
// This function moves the camera to the location of the
    player
[&](LambdaScript *ref, float dt){
    std::shared_ptr<Transform> t = ref->parent->get_component<
    Transform>();
    scene->active_camera->set_pos(t->get_position());
}
        );
```

2.3 Writing your own component

Every component inherits from the base class `Component`[1]. For your own component, you have to override two virtual methods.

First, you should start by creating some files[2].

- A .h file inside the inc/components/ folder that looks something like this.

```
// inc/components/mycomponent.h
#pragma once

class MyComponent : public Component{
    public:
        MyComponent(){}         // Every component needs and
    empty constructor for now.

        virtual void start();   // This is called only once
    after adding this component to a game object.
```

[1] Defined in inc/components/component.h

[2] Examples to these files can also be found under inc/components/template.h and src/components/template.cpp

```
        virtual void update(float dt); // Called every frame, before the physics and render steps.
};
```

- A .cpp file inside the src/components/ folder that looks something like this.

```
// src/components/mycomponent.cpp
#include <components/mycomponent.h>
#include <input.h>

void MyComponent::start()
{
    std::cout << "MyComponent added" << std::endl;
}

void PlayerLook::update(float dt){
    if(Input::is_key_down(KEY_E)) std::cout << "Button down << std::endl;
}
```

Then, you have to add your component to the component list by including the header and adding an entry to FOREACH_COMPONENT_TYPE macro.

```
// inc/components/componentlist.h
#pragma once
#include <components/collider.h>
#include <components/model.h>
#include <components/rigidbody.h>
#include <components/transform.h>
#include <components/camera.h>
#include <components/lambdascript.h>
#include <components/mycomponent.h>// <---- Include your component here

#define FOREACH_COMPONENT_TYPE(COMPONENT_TYPE) \
    COMPONENT_TYPE(Transform)                  \
    COMPONENT_TYPE(Model)                      \
    COMPONENT_TYPE(RigidBody)                  \
    COMPONENT_TYPE(Collider)                   \
    COMPONENT_TYPE(LambdaScript)               \
    COMPONENT_TYPE(Camera)                     \
    COMPONENT_TYPE(MyComponent) \ // <---- Add your component here
```

After this step, you can use your component as you want. Examples to custom components can be found under `src/components/fps_example/` and `inc/components/fps_example/` folder.