Tools Programming

Engine Tools

Class 9

Engine importer

- We need to fill ParseBin method
  - Define Chunk struct (id/size)
  - Define header struct
- Identify chunks of data by using magicId
  - Use fread function to read chunks of bytes
  - fill bytes data into vertices, normals and uvs vectors.
- Modify createGeometryFromFile method in graphic system to:
  - Allow reading .obj files (already done)
  - Allow reading .mesh files (to-do)

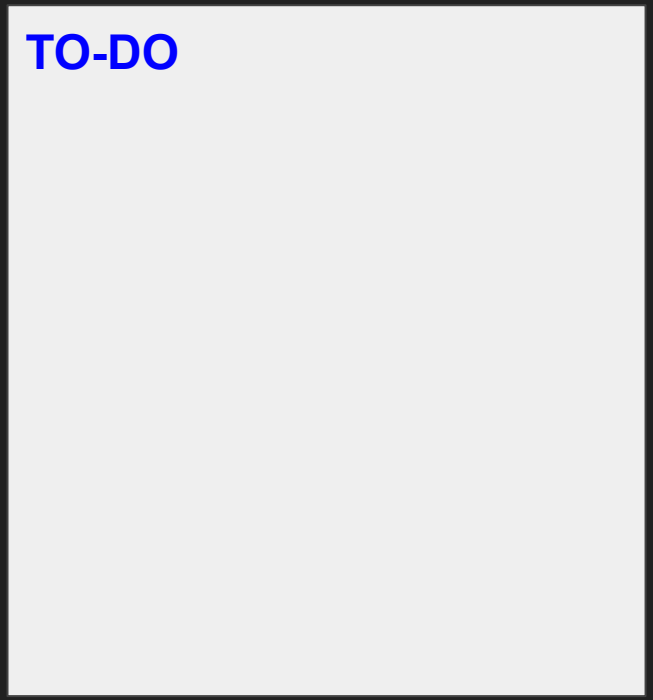
Custom Components

- We can create custom components in maxscript by
  - Creating a new attribute definition in mvd_components
  - Creating parameters and rollout definitions in mvd_components
  - Creating exporter/importer at 3DSMax and the engine parsing files

- We can create custom components in our engine by:
  - Creating a parser to import our custom component from the scene files
  - Declaring the component in Components.cpp file nad adding it to the stack
  - Declaring and implementing its functionality in a custom header and cpp files

TO-DO

- Define a custom component that:
  - Can be exported from 3DSMax to the engine
  - Must contain at least two attributes
  - Must have a current functionality in the engine
  - Engine must be able to read this component on scene load.
  - Engine must do an “update” function in this component



Updating components

- It’s oftly good to have functions that are common in all existent components
  - Declare the function update in components class.
  - Implement update template in EntityComponentStore
  - Implement update method for each custom component.
  - Override update function in custom component class.

- To keep some order, create folders for each of the new custom components that we will be creating.

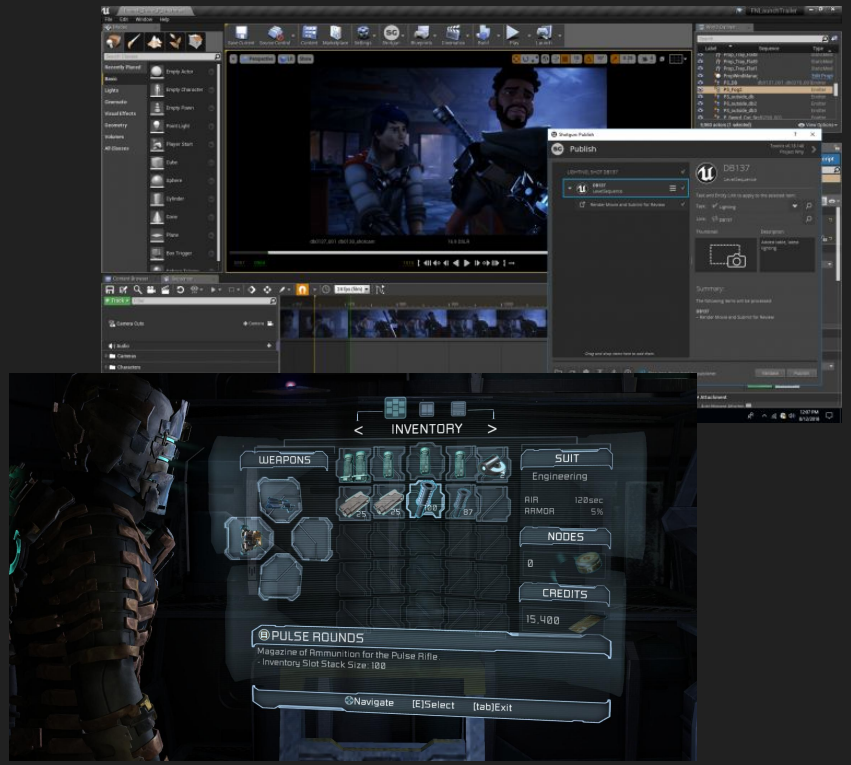
```
template<std::size_t I = 0, typename... Tp>
inline typename std::enable_if < I < sizeof...(Tp), void>::type
    updateComponents(std::tuple<Tp...>& t, float dt)
{
    for (auto& c : std::get<I>(t))
        c.update(dt);
}


//returns id of entity
void update(float dt) {

    auto& meshes = getAllComponents<Mesh>();
    auto& lights = getAllComponents<Light>();
    auto& colliders = getAllComponents<Collider>();
    auto& rotators = getAllComponents<Rotator>();

    for (auto& light : lights) light.update(dt);
    for (auto& col : colliders) col.update(dt);
    for (auto& rot : rotators) rot.update(dt);
}
```

Engine Interface

- We need to distinguish between the videogame interface and the engine interface. These interfaces implementation and layout is totally different between them.

- Engine interface is fully integrated within the engine and is not present in the final videogame build. Normally done with third party software

- Videogame interface is present in the final build and is made by planar geometry under an orthographic projection (normally)



ImGUI

- Graphical and open source interface library.
- Used to create debug and visualization tools.
- Fast implementation and easy to use
- Simply include the header and cpp files provided by the ImGUI repository.

ImGUI

- Installing ImGUI in our project:
  - No build needed, include the necessary files provided by the source repository
  - Include necessary headers in *includes.h*
  - *Add the files to imGUI filter within visual studio*

- Deploying ImGUI
  - Declare a ImGUI context in main.cpp
  - Declare the cleanup after game delete in main.cpp

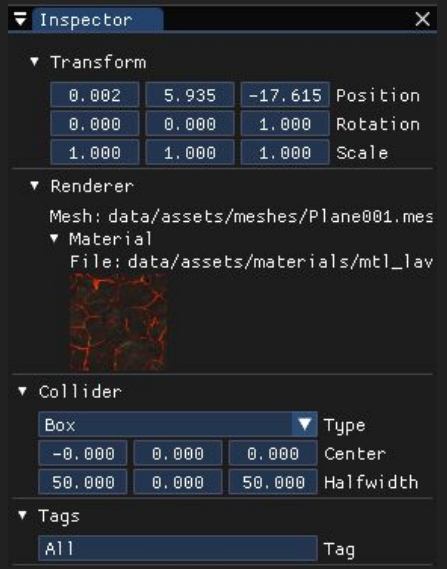
Building a hierarchy

- We need to build a complete independent window
- We need to create two functions to display scene data
  - UpdateHierarchy: to iterate through all entities (parent - children)
  - RenderNode: to display each node recursively and its associated children

- How to:
  - Use Begin and End to open a new window in the hierarchy
  - Use TreeNode class to create a hierarchy within the window
  - Disable control system while interacting with the engine UI.

TO-DO

- Display each component attributes in the hierarchy panel on object selection:
  - Create a debugrender method in components class
  - Override this method in all existent components
  - Show each component information in the hierarchy panel when unfolded
  - use unity inspector panel as reference!



Engine Editor Structure

- Under tools folder three new files
- Editor System:
  - Holds all information related to the editor UI and functionality
  - Holds the different modules present in the editor such as, console, graph editor…
  - Updates each window frame: hierarchy, inspector, project, scene…

- Console module:
  - Keeps a record on each command entered
  - Given an input, buildcommand method handles the answers

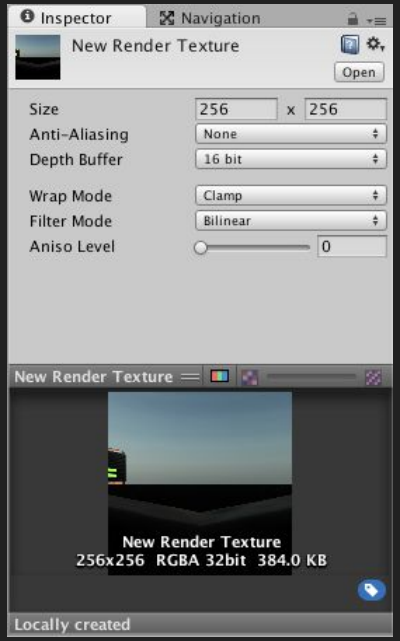
- Editor utils:
  - Mostly to read/write files and directories

- EditorGraph module:
  - Internal plugin to display blueprints
  - Unfinished..

Render To Texture

- In any engine, multiple cameras are used for different purposes:
  - Rendering the scene view
  - Rendering the game view
  - Rendering the UI
  - …

- We need to display this camera renders in different places.
- We need to save them as render targets (RTT)
- TO-DO: Display the scene view in an independent imGUI window to emulate Unity displays.



Developer Console

- A developer console is an input/output system by the engine
  - Engine consoles: To display engine errors or feedback (mostly)
  - Game consoles: used by dev/modders and even players to tweak configurations. Some game consoles are blocked by the devs once the game is launched.

- Why to use a developer console
  - To change settings very fast (Allow customization)
  - To display errors or other editor messages
  - To display engine feedback

Console: Command variables

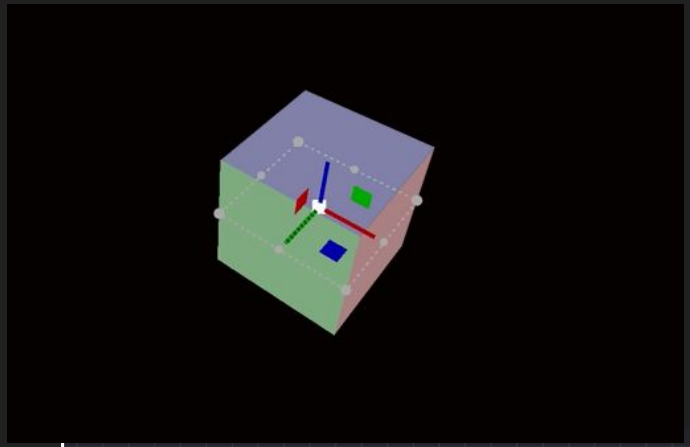
- Command variables are used to help the developer execute actions using the developer console to automate functionalities
  - Normally introduced with a “/” symbol to identify them
  - Normally each command belongs to a category and each category has a prefix.

- TO-DO: Declare a new command variable and describe its functionality

Link1: https://nwnlexicon.com/index.php?title=Category:Beginning_Scripting
Link2: https://scripts.zeroy.com/

More Tools

- Gizmos:
  - ImGUIzmo: Built in imgui gizmo library https://github.com/CedricGuillemet/ImGuizmo
  - We can also built our own (creativity)
    - Axis with colliders
    - Logic depending on selection
- Graph editors:
  - New programming paradigm
  - Many different ImGUI plugins presen https://github.com/ocornut/imgui/issues/306
- Timeline
  - Represent events in time
  - Many plugins present in ImGUI



Load/Save scenes

- We can save our engine editor changes.
  - Reading and keeping track of every change made
  - Overwriting the current scene json with the changes made

- How-to Implementation:
  - Implement a save method per component
  - Implement an scene save button
  - Once button clicked, for each component present call its save method
  - Note: If component hasn’t changed, do not overwrite information.

Final Deliver

- Limit date [14/01/2020 – 23:59:59]
- The student must provide a .zip file (or github repo) with the engine and tools files required by the app. Readme and map design image must be included.
- Content must be sent to alberto.s@salle.url.edu before limit date.
- Mail header must be [MVDTools-MyFullName]
- Work must be done individually by each student.
- Provide a readme (.txt file) with the description of the work done.
- The plugins must execute without errors.
- The engine must execute without errors.