Tools Programming

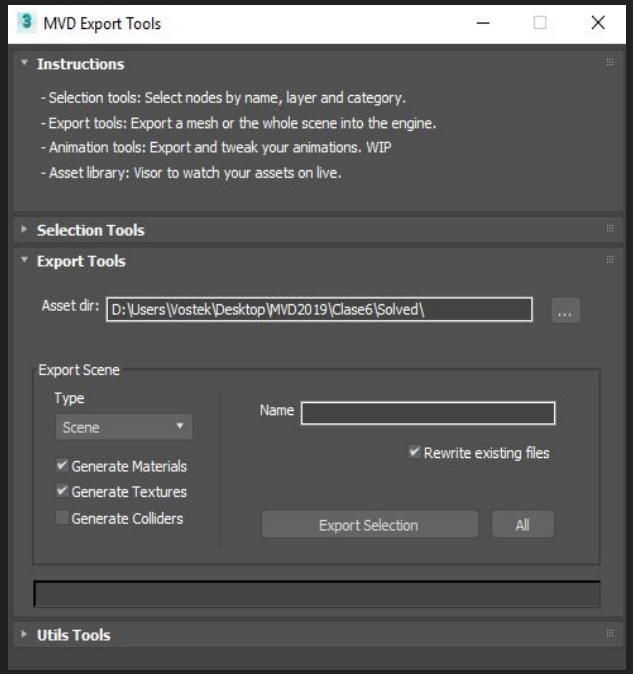
Asset integration pipeline 2

Class 7

Index

Sample

- Update the UI to provide exporting tools for:
  - Scene export
  - Mesh export
  - Prefab export (next step)
  - Curve
- Export can be done with:
  - All scene items
  - Only selected items.
- We can force
  - Copy materials
  - Copy textures



JSON Library

- Json library allow us to:
- Serialize data into a string formatted data like JSON
  - Easy to read
  - Easy to manipulate
  - Easy to interpret

- The core functionality exposed by:
  - jwrite.add key value (to write a key value pair)
  - Library reads and write components

- **Add support to print data into json formatted file**

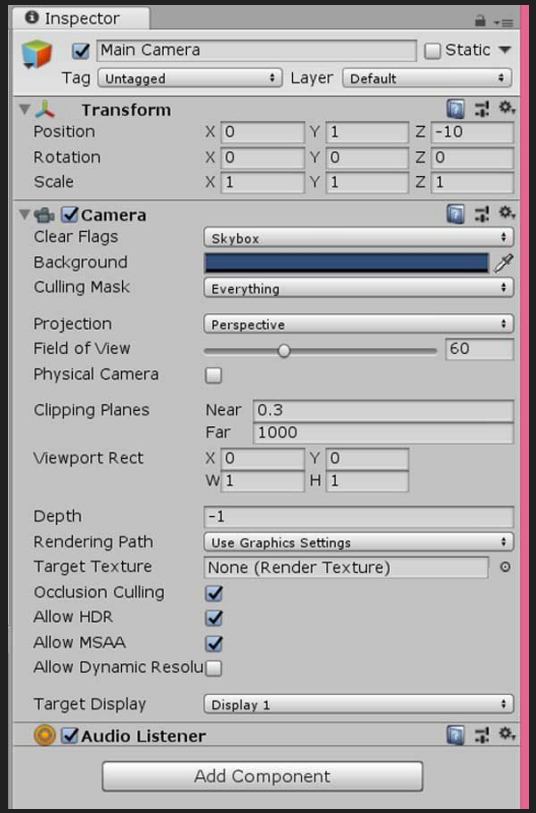
Scene format

- Scene is composed by objects
- Objects are composed by components
  - Transform component (default)
  - Collider component
  - Light component
  - ….


```
{
  "scene":"test",
  "entities":[
    {
      "name":"Teapot001",
      "transform":{
        "translation":[
          0.321126,
          3.48027,
          -1.52127e-07
        ],
        "rotation":[
          0.0,
          0.0,
          0.0
        ],
        "scale":[
          1.0,
          1.0,
          1.0
        ]
      },
      "render":{
        "mesh":"data/assets/meshes/Teapot001.mesh",
        "materials":[
          "data/assets/materials/mtl_default.mtl"
        ]
      },
      "tags":[
        "default"
      ]
    }
  ]
}
```

Scene format

- We are following Unity format!
- We are internally using JSON as it is more readable.
- Unity internally also uses a similar structure but encrypted.
- Unity uses components as property “categories” for each of the objects.



Sample

- To export the scene we must export every entity by:
  - Export its mesh by using geometry exporting class
  - Export its maps by using material exporting class
  - Write its component to json file
    - Name component
    - Parent component (hierarchy)
    - Tag component (category)
    - Export prefab (if its the case)
    - Export transform component
    - Export Renderer component
  - We must distinguish between:
    - Scene, Prefab
    - Curve, Mesh

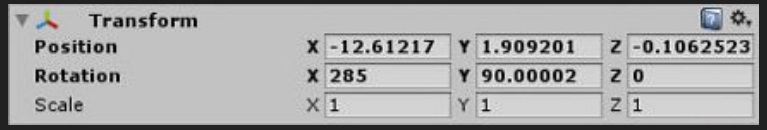
Exporting Components

- ExportName
  - Name is an internal component in Unity
  - It is a must component in all entities

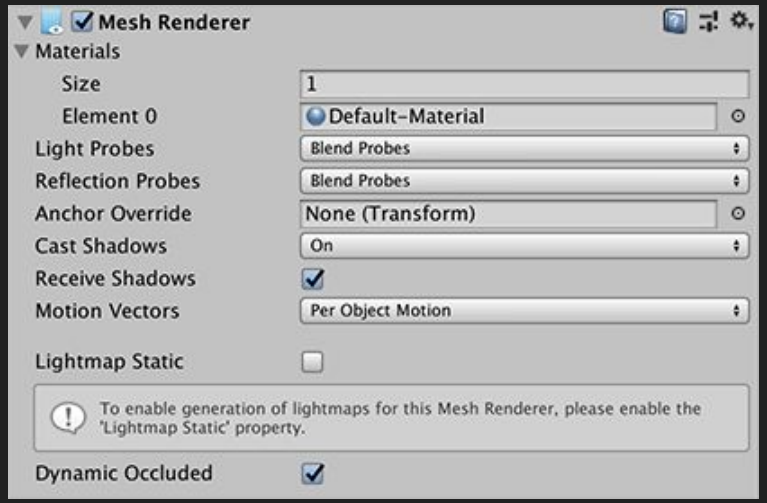
- ExportParent
  - Some objects might have a parent/children relationship
  - We link them with a component named parent

Exporting Components

- Transform component
  - Translation
  - Rotation
  - Scale

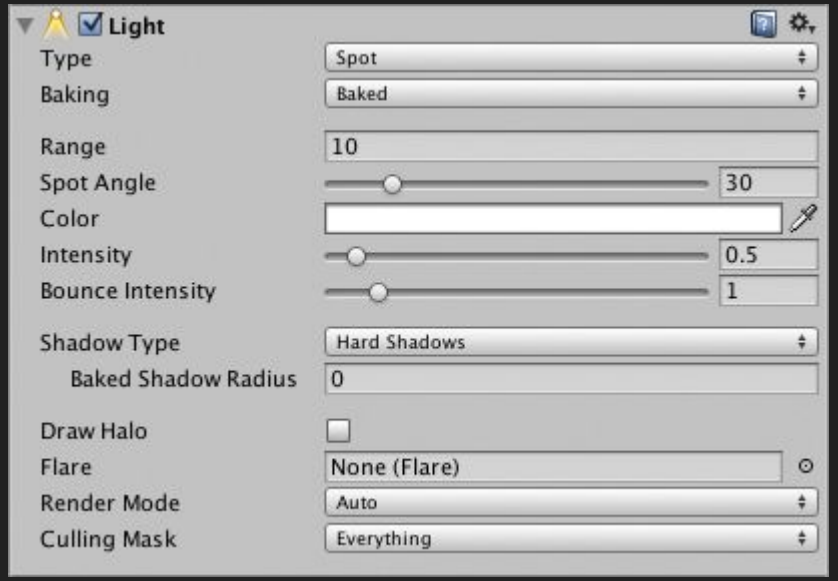


- Render Component
  - Mesh
  - Materials
  - Other mesh properties



Exporting Components

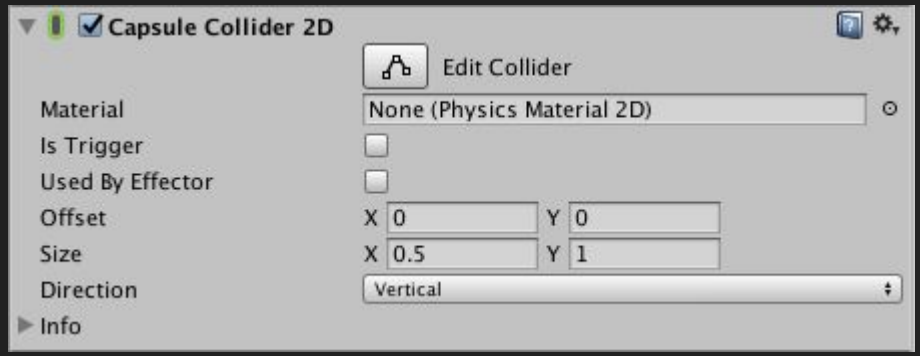
- Light
  - Type
  - Projection type
  - range
  - Ortosize/Fov
  - Attenuation Start/End
  - Intensity
  - Color
  - Shadows On/Off



Exporting Components

- Collider
  - Collider type
  - Material
  - Center
  - Size
  - IsTrigger
  - Dynamic
  - Gravity

- Spline
  - Type
  - Knots list



Exporting the scene

- Using the following methods export an scene, mesh, prefab:
  - *ExportWorld*
  - *ExportMesh*
  - *ExportPrefab*

- Use *ExportItem* method to branch between different options
- Use *ExportEntity* and *ExportEntityPrefab* to export entities and prefabs

TO-DO

- Create a simple scene with 3DSMax.
- Add a prefab to the scene.
- Export the scene using the previous tools created.
- Use json formatter to ensure that the scene generated is correct.

Project Structure

- Our Maxscript tools folder must be placed within our engine folder
  - To link all data paths of our tools to the engine
  - To link source control of engine and tools together
  - To be easy to manually access if needed

- Engine Folder
  - Maxscript folder (Plugins folder)
  - Data folder (assets related)
  - Include (libraries)
  - Src (code)
  - Visual studio (vs files)
  - Exe and other.

Importing

- Once the json scene file is generated we need to read it by the engine
- We need to parse all the information and create all entities in the engine
- To do so we need:
  - A JSON parse tool
  - Have an scene that can be identified by an id or name
  - Have all the necessary resources needed by the scene within the assets folder
  - Add the necessary code to parse all the data with the json tool.

RapidJSON Library

- We will use RapidJSON library to import our scene into the engine
- Library can be used by including the following headers:
  - #include "rapidjson/document.h"
  - #include "rapidjson/istreamwrapper.h"

- Works with the following methods:
  - Document.Parse: to parse the file
  - HasMember: To check if given node has an attribute of the given type
  - GetString(), GetArray(): to parse vectors and strings
  - HasParseError: to check if string is a valid json.

Auxiliar methods

- EntityComponentStore
  - GetEntityByName: retrieve entity matching string provided
  - GetComponentFromEntity: retrieve component matching string provided

- Parser file
  - ParseScene: scene with json as parameter, parses every entity on it
  - ParseEntity: given the selected entity parses its component and creates the new entity with initialized data.
  - ParseBin: parses all the binary meshes provided by the scene

TO-DO

- Finish the scene importer in the engine
- In Parsers.cpp implement the following logic:
  - Finish entity parsing on ParseScene function
  - Finish entity parsing components on ParseEntity method:
    - Parse prefabs
    - Parse transform component
    - Parse render component
    - Parse light
  - Generate the given entity in the world with all his listed components

- Remember to load an scene on game.cpp with the following line
  - Parsers::parseScene("data/assets/scenes/scene_test.scene", graphics_system_);