FURY: GLTF Integration

GSoC 2022 Proposal

Organization: Python Software Foundation

Sub Organization: FURY

Project Length: Full-Time Project (350 Hours)

Contents

About Me

Personal Info

- Name: Shivam Sahu
- Country: India
- TimeZone: India (UTC +5:30 hours, EST +9:30)
- Github: Xtanion
- Linkedin: Shivam Anand
- Email: anandshivam54321@gmail.com
- Discord: xtanion#0280

University Info

- University: Indian Institute of Technology, Roorkee
- Major: Production and Industrial Engineering
- Current year: 2nd year
- Degree: Bachelor of Technology (2024 expected graduation)

Introduction

I am a second-year student at IIT Roorkee, majoring in Production and Industrial Engineering (a 4-year course). I started programming in the first semester of my college. I explored many python frameworks such as Numpy, Flask, TensorFlow, Pytorch, PyGame, etc. I have worked on many projects in the field of Deep Learning, Android App development & Game Development. One of my group projects was to create an OCR to detect mathematical equations (under the surveillance of the Physics and Astronomy Club, IITR).

In order to understand computer graphics and OpenGL, I've gone through UCSC university lectures and OpenGL tutorials. I read a bit of documentation on VTK. I've read the documentation of FURY and tried to implement some demos/tutorials. I find FURY fascinating as it's very easy to use and versatile. I'm currently learning about Shaders and Animations to add good contributions in FURY.

Programming Skills

Programming Languages and Frameworks

- Fluent in Python, C, and Kotlin
- Moderately experienced in C++, Java, and SQL
- Experienced with various python frameworks (Numpy, Tensorflow, PyGame).

Development Environment

- Ubuntu 21.10 (primarily used) dual booted with Windows 11
- Visual Studio Code and Pycharm as primary text editors
- Good grasp on the concepts of Git
- Moderately experienced with Blender & CAD software (AutoCAD).

Code Contributions

Below is the list of PRs that I've worked on FURY codebase -

- #520. Added rotation along the axes in Solar System Animation example **(merged)**

  An **Enhancement** PR. Previously the planets in the solar system example, didn't have rotation along the axes. Added rotation relative to the earth's rotation period.

- #525. Implemented VtkBillboardtextActor **(opened)**

  Implemented **VtkOpenGlBillboardTextActor3d** that always faces the camera no matter the orientation. It aimed to resolve #465.

- #533. Sphere actor uses repeat_primitive by default **(opened)**

  I modified `prim_sphere` to generate a sphere using `phi` and `theta` parameters and sphere actor to use primitive by default. It aimed to resolve #528.

- #547. Cone actor uses repeat_primitive by default **(merged)**

  Implemented the **prim_cone** primitive. Modified cone actor to use it by default and added unit tests for actor and primitives, respectively. It aimed to resolve #529. It was a **New Feature** PR.

- #556. Updated code of viz_network_animated to use fury.utils **(merged)**

  A **documentation** PR, In which I refreshed the code of demo `viz_network_animated` to use `fury.utils` function. It resolves #553

- #559. Added simulation for Tesseract **(merged)**

  I made a simulation of the Tesseract (4-D extension of a cube), rotated and projected it into 3D so that it can be visualized by scene. Added tutorial for the same. It was a **documentation** PR.

Some issues that I've raised -

- #526. Text Justification in VtkTextActor3D

  Text justification of vtkTextActor3D seems to be working fine. There's a **_update_user_matrix()** method in the text3d in fury; with or without this method, the text actor's justifications were working in the same way

- The direction of actor produced by repeat_primitve is not the same as vtkActor

  `direction` parameters of a vtkActor and primitive work in different ways. Hence, actors made from primitives get different orientations.

Project Information

Abstract

I will be working on integrating the **glTF** (Graphics Language Transmission Format) file format to FURY. The glTF format minimizes the size of 3D assets and unpacking time. The 3D data object can be directly used by any common graphics APIs, so there won't be any need to decode or pre-process the 3D data. By the completion, We shall be able to load/export and animate `.gltf` format.

As part of my **Stretch Goals,** I would like to work on expanding fury.io to integrate with GLB formats as well.

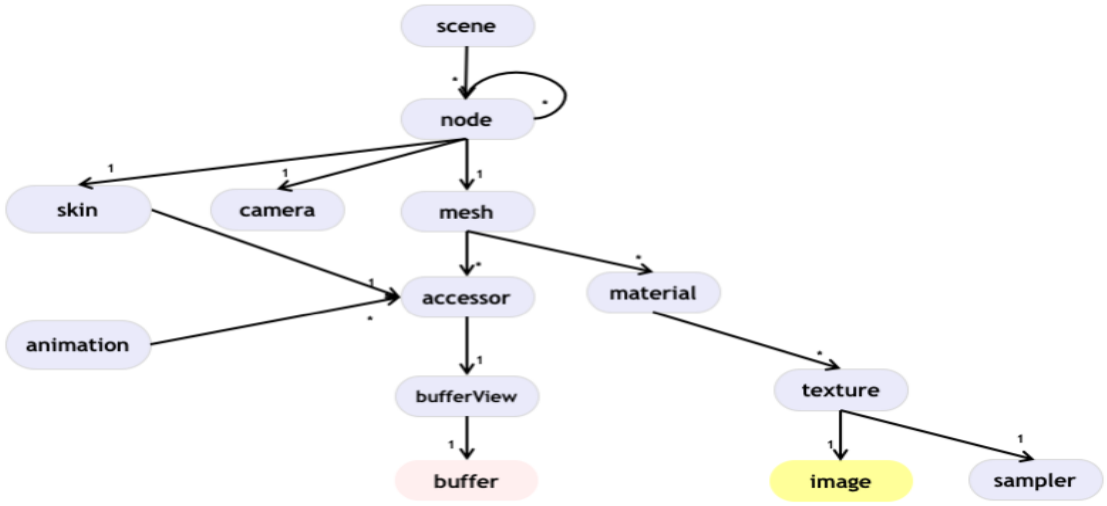
Detailed Description

The glTF format is used to store 3D data in a suitable form for all runtime applications. glTF allows us to store geometry data and scene data while keeping the file size compact. Multiple 3D file formats have already been integrated into FURY (OBJ, PLY, STL). However, glTF has its advantages of compact size and faster load times.

Extracting data from glTF

The .gltf file contains scene structure as JSON. The scene data can be extracted using the built-in `json` library in Python.

The JSON file contains the scene structure which is given by the hierarchy of nodes that define the scene graph.



(glTF 2.0 structure, source: glTF 2.0 specification)

The **Node** contains transformations (Translation, Rotation, Scale) multiplied into T*R*S to give the transformation matrix. It also contains **Meshes**, **Camera**, or **Skin** (defines mesh transformations) instances.

Meshes have `mesh.primitive,` which contains references to the accessor, UV texture coordinates, Normals, Vertex Colors, Weights, and Joints for skinning.

```
"meshes": [
    {
        "primitives": [
            {
                "attributes": {
                    "NORMAL": 23,
                    "POSITION": 22,
                    "TANGENT": 24,
                    "TEXCOORD_0": 25
                },
                "indices": 21,
                "material": 3,
                "mode": 4
            }
        ]
    }
]
```

POSITION redirects us to the accessor; TEXCOORD_0 refers to UV coordinates for the first set.

An **accessor** defines a method to parse data from `buffers` using `bufferview` and `bufferOffset`.

```
"accessors" : [
  {
    "bufferView" : 0,
    "byteOffset" : 0,
    "componentType" : 5123,
    "count" : 3,
    "type" : "SCALAR",
    "max" : [ 2 ],
    "min" : [ 0 ]
  },
```

(In this accessor, component type is 5123(Float), so each component is 4 bytes. Type is SCALAR; hence the size of each element is 4 bytes (4*1).)

A `bufferView` represents a slice of data in one buffer. It contains `buffer`, `byteOffset` & `byteLength`.

```
"bufferViews" : [
  {
    "buffer" : 0,
    "byteOffset" : 0,
    "byteLength" : 6,
    "target" : 34963
  },
```

(this bufferView refers to the 0th buffer, the byteOffset is 0 and byteLength is 6. So, this bufferView is referring to the 0-6th row in the buffers)

A `buffer` represents a block of raw binary data. Data is referred to as buffer by `uri`. A `uri` may contain the path to an external `.bin` file or it could be a data URI that is encoded directly into JSON. It can be decoded using the `base64` module and it'll provide us with Vertices, Normals, Color Values & UV data.

Materials data is again present in JSON format. Materials are assigned to meshes.

```
"materials" : [
  {
    "pbrMetallicRoughness": {
      "baseColorFactor": [ 1.000, 0.766, 0.336, 1.0 ],
      "metallicFactor": 0.5,
      "roughnessFactor": 0.1
    }
  }
],
```

(Example of material in glTF 2.0.)

Textures refer to images that contain `uri` to the image that we'll be using as texture. It can be extracted as a relative file path. We can create a function using `vtkTexture` to map the texture to the `actor`, then using `setTCoords` (Set Texture Coordinates) to map the UV coordinates that we got from the buffer. VTK provides various ways to set textures, for eg: vtkProjectedTexture.

```
"textures": [
  {
    "source": 0,
    "sampler": 0
  }
],
"images": [
  {
    "uri": "testTexture.png"
  }
],
```

(Here, the texture source refers to the 0th object in images).

The material is applied after the texture is applied to the `actor`.

Translating data from glTF to FURY

Steps involved in visualizing a basic glTF (without animations) from the data:

1. Creating actors from the verts faces data (& if present, UV coordinates) that we got from buffers. From the vertices and faces data, we'll create a simple primitive model and then convert it into an actor (We can use `utils.repeat_sources`).

2. Applying the node transformation matrices and updating vertices of the actor.
3. Get the texture image using the `io` module. We can use `vtkTexture(`or `fury.texture)` with UVTexture mapping to do so. Here are some examples: example1, example2.
4. Identify the material type (eg: pbrMetallic) and set it to actors using `fury.material`. Update actor.
5. Similarly the lights and camera can be set to FURY using the built-in `scene.camera` module.

Exporting a simple FURY model as glTF:

1. Get the primitives data (vertices, triangles, color values, UV list, etc.) from the scene and create nodes for each actor. Convert the primitive data into binary and store them in buffers (convert to base-64 if the user wants to, else store them in a separate .`bin` file).
2. Get Material and texture data, and store them into the mesh of the respective node. Store the camera data into a `node`.

Animations

glTF supports Node transforms, Morph Targets weights, and skinned animations via key-frame animations.

The animation data describes how the T, R & S properties of a node change over time. An `animations` object consists of two elements: `samplers` and `channels`. `samplers` refer to `accessor` objects, and it also defines how the data should be interpolated into keyframes. `animations.channel` connects the `sampler` with the node. We can get the target node from `target.node` and `target.path` defines the animation property (T, R, or S).

```
"animations": [
  {
    "samplers" : [
      {
        "input" : 2,
        "interpolation" : "LINEAR",
        "output" : 3
      }
    ],
    "channels" : [ {
      "sampler" : 0,
      "target" : {
        "node" : 0,
        "path" : "rotation"
      }
    } ]
  }
],
```

(Here, the 0th node is the target node and the animation type is rotation. The interpolation type is LINEAR, input 2 and output 3 refer to time and animation accessors.)

In FURY, we don't have a proper way to animate using keyframes. There's a separate GSoC project for the implementation of keyframe animations. However, this could be an alternative approach to deal with the animations:

In FURY we can use `timer_callbacks` to update and animate the scene. We can replicate the effect of keyframe animations with existing timer_callbacks explained by the following pseudo-code:

```
num_frames = accessor['count']  # frame counts FPS
duration = 1 / num_frames  # the refresh duration of timer_callback
duration = duration / n  # dividing the duration into n (arbitrary)
parts

counter = itertools.count()
def timer_callback(_obj, _):
    cnt = next(counter)
    if cnt % n == 0:
        tra_data = get_data_from_accessor(anim_acc,  'translation')
        rot_data = get_data_from_buffer(anim_acc, 'rotation')
        scl_data = get_data_from_accessor(anim_acc, 'scale')
    else:
        interp_type = sampler['interpolation']
```

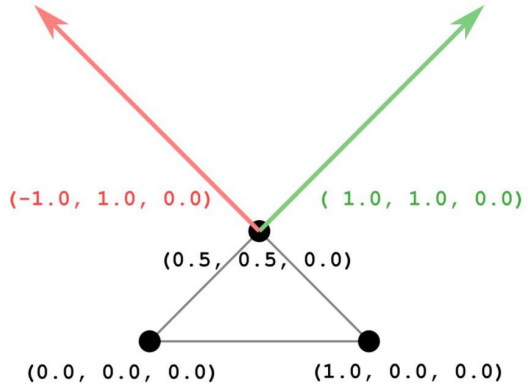
```
        interpolationValue = (currentTime - previousTime) /
(nextTime - previousTime)
        if interp_type == 'STEP':
            pass
        elif interp_type == 'LINEAR':
            tra_data, rot_data, scl_data = interp_lerp()
        elif interp_type == 'CUBICSPLINE':
            tra_data, rot_data, scl_data = interp_cubicspline()

    # from the above data we can calculate changes in node's
    # T, R, & S properties.
```

Morph Targets: Morphing is a process of changing from one model to another. Morph Targets stores displacements or differences for certain mesh attribute that is added to the original mesh at runtime to animate -

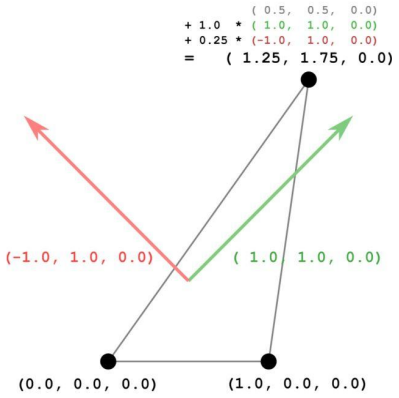
```
"meshes":[
  {
    "primitives":[
      {
        "attributes":{
          "POSITION":1
        },
        "targets":[
          {
            "POSITION":2
          },
          {
            "POSITION":3
          }
        ],
        "indices":0
      }
    ],
    "weights":[
      0.5,
      0.5
    ]
  }
],
```

an object. We can determine the Rendered primitive position using the following pseudo-code:

```
renderedPrimitive.POSITION = primitive.POSITION +
  weights[0] * primitive.targets[0].POSITION +
  weights[1] * primitive.targets[1].POSITION;
```



Time	Weights
0.0	0.0, 0.0
1.0	0.0, 1.0
2.0	1.0, 1.0
3.0	1.0, 0.0
4.0	0.0, 0.0



(Here, the position is computed at t=1.25 seconds. source: glTF 2.0 specifications)

From the `animations`, we get the data that affects `weights` for a given keyframe and timestamp. Weights are interpolated linearly using that data and applied to the morph target displacement. glTF currently supports morphing of vertex position, normals & tangent data.

glTF 2.0 only supports animating node transforms, skinning, and Morph Targets weights. It doesn't have support for animating material colors and texture transform matrices (source).

Implementing Skeletal Animation (Skinning)

Vertex Skinning allows the vertices of a mesh to be influenced by the bones of the skeleton. Skinning glTF data is stored in nodes, meshes, and the skin object. I'll be implementing skeletal animations as part of glTF animations.

A node refers to the index of the skin object. A skin object contains an array of joints that defines the skeleton (bones) hierarchy, that is the joints that are to be affected by the node transformation. Each vertex also contains a **weights** array (range: [0, 1], contains up to 4 values) that defines the effect of the movement of bones to the vertex. Skin object also gives us the reference to **inverseBindMatrices** for each joint. Using the information, animated position can be calculated as:

```
>>> Panimated = Pmesh * InverseBindMatrixjoint* MBM
>>> Panimated = Pmesh * SkinningMatrix (MBM = Transformation Matrix)
```

Applying the transformation matrix to the mesh in a loop will create an animation effect.

Creating Demos & Adding Tests

After writing code for complete integration of glTF and adding Animations, I'll add demos to show how they can use the new glTF importer/exporter in FURY. I'll also add unit tests for the newly implemented glTF reader and writer.

Stretch Goals

Binary glTF (GLB) Integration

GLB is like glTF, but unlike glTF, it contains everything (JSON, buffers, and Image data) as a single binary blob. It can still refer to external resources (e.g., Texture for multiple objects can be referenced by uri).

After complete integration of glTF (load/ animate/ and adding demos for the same), I'm planning to expand `io` with GLB format as well.

Project Timeline

Period	Goals / Milestones
	Community Bonding period May 20, 2022 - June 12, 2022
Week 1-3 May 20 - June 12,	• Getting to know about the community, mentors, and the admins. • Reading up glTF standards and other related projects (e.g., panda3D glTF). • Discuss the goals for the mentioned features. • Discuss the order of Implementation. • Review and merge any pull requests that could aid the project and otherwise. • Read the binary data and convert it into NumPy arrays. • Parse the glTF file and create objects to save them into memory.
	Phase 1 Coding June 13, 2022 - July 25, 2022
Week 4 June 13 - June 20	• Load meshes (without textures/materials/animations) from glTF and display them to the scene. • Create functions for applying node transformations.

Week 5 June 21 - June 28	• Load materials and texture data. • Create a function to apply textures using vtkTexture (or suggested by the mentor).
Week 6 June 29 - July 6	• Apply materials to the mesh using built-in methods.
Week 7 July 7 - July 14	• Add lights, mentioned camera position. • Fix any bugs in Importer.
Week 8, 9 July 15 - July 24	• Parse the `animations` data from `buffers`. • Creating callback function to mimic the keyframe animation.
Phase 1 Evaluation July 25 - July 29	
Week 10 July 25 - Aug 1	• Write functions for all supported interpolation types. We can use `scipy` interpolation and spatial transformations. • Apply interpolation for intermediate frames.
Week 11 Aug 2 - Aug 9	• Create functions for simple node transformation animations. • Perform bug fixes
Week 12, 13 Aug 10 - Aug 25	• Start loading the morph data. • Interpolate weights of morph targets on each callback to perform morph animation. • Update vertices of the mesh.
Week 14 Aug 26 - Sep 2	• Perform bug fixes in Morphing Animations • Create bones hierarchy (bind pose) for the skeletal animation. • Assign vertices to bone indices and their weights.
Week 15 Sep 3 - Sep 10	• Load skinning information (inverseBindMatrices, weights, etc).
Week 16, 17 Sep 10 - Sep 25	• Calculate transformation matrices for the hierarchy of bones • Calculate skinning matrices. • Apply skinning matrix and weights to the vertices
Week 18 Sep 26 - Oct 3	• Apply skinning matrix and weights to the vertices • Transform the vertices.

Week 19 Oct 4 - Oct 11	• Debug the skeletal animations code. • Add unit tests for loading the glTF.
Week 20 Oct 12 - Oct 19	• Add unit tests for animations, morphs & textures.
Week 21 Oct 20 - Oct 27	• Work on stretch goals (if time permits). • Try to integrate the `.glb` format into `fury.io`
Week 22, 23 Oct 28 - Nov 13	• Debug the code extensively for loading, visualizing, and animation. • Prepare tutorial/demos/videos for newly implemented features.
Final Evaluations Nov 14 - Nov 21	

Commitments and Availability

---

1. My college End Term Examinations are from 22 April to 2 May & College's summer vacations will continue for the entire month of May, June, till July 22. Therefore there won't be any classes during the First Coding Period & I'll be able to invest at least 40-45 hrs a week.
2. My college will reopen on July 25. During this time, I may have offline classes in the evening for 2-3 hours daily; during this time, I'll be able to work 40-45 hrs a week.
3. I have no plans of any personal vacations nor of any travel otherwise.
4. Typical working hours (in EST): 11:30 PM to 2:30 AM, 4:30 AM to 7:30 AM, 12:00 PM to 3:00 PM.

5. I’m willing to reschedule and re-plan the timings as per my mentor’s / other team member’s availability/requirements.
6. I’m only applying to FURY in GSoC 2022.