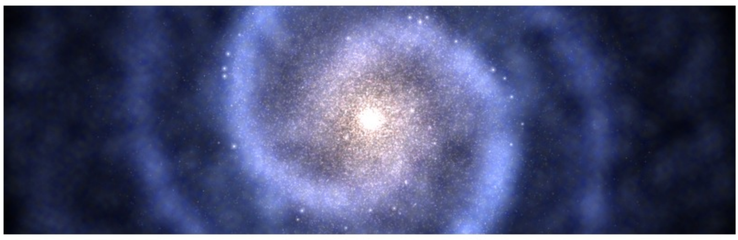
Memory-aware Array: Glumpy

This lesson briefly discusses a very commonly used Library in Python called "GlumPy".

Glumpy is an OpenGL-based interactive visualization library in Python whose goal is to make it easy to create fast, scalable, beautiful, interactive and dynamic visualizations.



Simulation of a spiral galaxy using the density wave theory.

Tiger display using collections and 2 GL calls

Glumpy is based on tight and seamless integration with NumPy arrays. This means you can manipulate GPU data as you would with regular NumPy arrays and glumpy will take care of the rest. But an example is worth a thousand words:

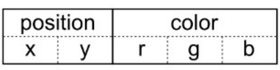
```
from glumpy import gloo

dtype = [("position", np.float32, 2),  # x,y
         ("color",    np.float32, 3)]  # r,g,b
V = np.zeros((3,3),dtype).view(gloo.VertexBuffer)
V["position"][0,0] = 0.0, 0.0
V["position"][1,1] = 0.0, 0.0
```

`V` is a `VertexBuffer` which is both a GPUData and a NumPy array. When `V` is modified, GlumPy takes care of computing the smallest contiguous block of dirty memory since it was last uploaded to GPU memory. When this buffer is to be used on the GPU, GlumPy takes care of uploading the “dirty” area at the very last moment. This means that if you never use V, nothing will be ever uploaded to the GPU!

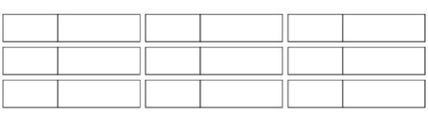
> **Note:** *Dirty memory is a memory that represents data on disk that has been changed but has not yet been written out to disk.*

In the case above, the last computed “dirty” area is made of 88 bytes starting at offset 0 as illustrated below:

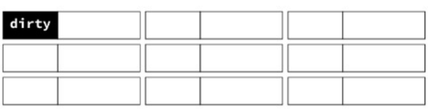
```
dtype = [("position", np.float32, 2),
         ("color",    np.float32, 3)]
```



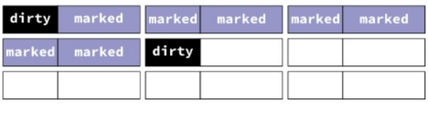

```
V = np.zeros((3,3),dtype)
V = V.view(gloo.VertexBuffer)
```



```
V["position"][0,0] = 0.0, 0.0
print (V.pending_data)
(0,8)
```




```
V["position"][1,1] = 0.0, 0.0
print (V.pending_data)
(0,88)
```



> **Note:** When a buffer is created, it is marked as totally dirty, but for the sake of illustration, just pretend this is not the case here.

Glumpy will thus end up uploading 88 bytes while only 16 bytes have been actually modified. You might wonder if this optimal. Actually, most of the time it is, because uploading some data to a buffer requires a lot of operations on the GL side and each call has a fixed cost.