Real Time Interactive Deformer Rig Evaluations in Maya using GPUs

FunShing Sin
Blizzard Entertainment
fsin@blizzard.com

Parag Havaldar
Blizzard Entertainment
phavaldar@blizzard.com

Vinod Melapudi
Blizzard Entertainment
vmelapudi@blizzard.com

ABSTRACT

Maya has supported evaluating deformer nodes on GPU since 2016. However, such GPU support in Maya is limited to evaluating simple linear chains of deformer nodes. In feature productions, character rigs have a complex network of deformation chains resulting in most deformers being evaluated on CPU. Here we will detail such architectural limitations within Maya. Then we present our approach that overcomes these limitations to fully evaluate deformation networks on GPU, which has enabled our rigs to perform over 50fps on GPU, compared to 5fps on CPU.

CCS CONCEPTS

• **Computer systems organization → Real-time systems**;

KEYWORDS

GPU, Deformers, Parallel Evaluation, Maya, Rigging, Animations

ACM Reference Format:

FunShing Sin, Parag Havaldar, and Vinod Melapudi. 2021. Real Time Interactive Deformer Rig Evaluations in Maya using GPUs. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Talks (SIGGRAPH '21 Talks), August 09-13, 2021.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3450623.3464666

1 INTRODUCTION

Deformers are computation nodes in Maya that are used to deform meshes and are essential building blocks in our character rigs. Skinning, Binding, Relaxing are some examples of deformers. A deformer node takes mesh(es) as input, modifies the vertex positions, and returns the deformed mesh(es) as output. The deformations are effected by parameters such as floating points (smoothIterations in deltaMesh), matrices (joints in skinCluster), and meshes (targetShapes in blendShape). We categorize deformers into two types depending on whether they take additional meshes as input in their computation:

Type I: Deformers affected by additional parameters in their computation but do not use additional meshes. e.g., skinCluster, deltaMush, relax.

Type II: Deformers affected by additional mesh(es). e.g., blendShape, wrap.

SIGGRAPH '21 Talks, August 09-13, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8373-8/21/08.
https://doi.org/10.1145/3450623.3464666

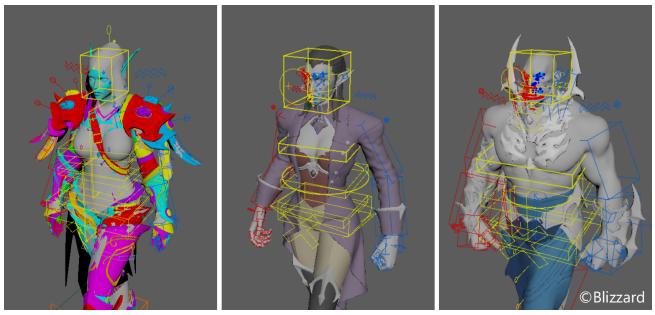

Figure 1: Production rigs, Sylvanas, Venthyr, Jailer.

In a production rig, character meshes are usually affected by multiple deformers. Given a chain of deformers, Maya is able to run all deformers on GPU if either of the following is true:

- the chain has Type I deformers only.
- the chain contains Type II deformers, and the mesh parameters are static/non-animating (e.g., Figure 2a).

Unfortunately, if a Type II deformer has mesh parameters that are non-static/animating (Figure 2b), Maya evaluates the upstream deformers (connecting to the mesh parameter) on CPU [1].

The rigs developed at Blizzard Animation heavily relies on such deformers. So this limitation significantly affects the performance of such rigs when animating. So even with availability of GPU implementations of those deformers, our test rig runs only at around 5fps.

2 SOLUTION FOR SMALL SCALE RIGS

While investigating the above-mentioned limitations, we observed that when an animated mesh is passed as a parameter to a deformer, that mesh does not get loaded on GPU. The reason for this is illustrated in Implementation 1. The GPU evaluate function accesses the animated meshB's data via the MDataBlock. Anything that accesses MDataBlock must be on CPU prior this evaluation. Our initial solution was to introduce a pass-through deformer upstream whose job is to initiate the loading of all meshes into GPU memory. The DAG is then modified to have deformers sequentially evaluated on GPU where each deformer accesses their needed mesh(es) via GPU device memory pointers. This is illustrated in Implementation 2 & 3 which ensures all computation on the GPU and the modified DAG is shown in Figure 2c. The performance gain using this method was not significant, improving from 5fps on CPU to 12fps on GPU. The main reason for this modest increase is because the individual meshes being deformed have vertex counts less than 2K which does not make significant use of GPU throughput.

[1] Autodesk has addressed this issue in Maya 2022, achieving similar speed up as the first solution discussed in section 2. We present further improvement in section 3.

3 SOLUTION FOR LARGE SCALE RIGS

For large scale rigs, combining all meshes into one mesh should yield a better throughput from GPU. Correspondingly the meshes were appended together sequentially to keep track of vertices and faces of individual meshes. The vertex range information (which decides which verts belong to various deforming submeshes) are passed along to each deformer. Each deformer in the modified DAG (Figure 2d) sequentially evaluates and deforms only the requisite vertices from the combined mesh. Another alteration that needs mentioning is the change in the usage of many skinClusters. Since the modified DAG allows only one skinCluster at the root of the deformation chain, we combine the binding information from all the skinClusters and create one super-skinCluster that affects the combined mesh. Artist workflows could be impacted by such mesh combinations. For instance, artists may toggle visibility of meshes for performance/workflow reasons. Such workflow needs can be reintroduced on a case-by-case basis in this combined setup – eg visibility could be now controlled by transparent shader, or even additional deformers that collapse the vertices to one point. Additionally, by combining meshes, we also benefited from avoiding many world-to-local space conversions. The performance gains using this method have been 10x as illustrated in Table 1.

4 CONCLUSION AND FUTURE WORK

The limitations of evaluating deformations completely on GPU within Maya, made our character rigs not interactively animatable for our artists. To overcome those limitations, we first detailed on how a PassThrough deformer, that loads dependent meshes on GPU, helped our deformations fully evaluate on GPU. And to further support larger scale rigs, we detailed our approach of combining the isolated mesh pieces and deforming them by vertex range. Our solution has shown significant speedup in rig evaluation. Our animators have tested and verified the interactivity they gain with the modified rigs. We are currently addressing the changes in our rigging process and the various proprietary deformers to benefit our future productions.

Table 1: Performance on Maya Parallel mode with i7-7820X and GTX 1080

	#Vertices	CPU (fps)	GPU (combined mesh, fps)
Sylvanas	198k	5	50
Venthyr	408k	3	36
Jailer	270k	2	30

ACKNOWLEDGMENTS

We thank Changyaw Wang, Shaik Sadiq, Martin Bisson, Martin De Lasa, Aloys Baillet, and Andy Lin.

```
DStatus TypeII::evaluate(MDataBlock& db,.,const MGPUDeformerData& inData,.){
   // a. get meshA vertices from GPU buffer (sPos)
   MGPUDeformerBuffer bufferMeshA = inData.getBuffer(sPositionsName());
   // b. get meshB vertices from MDataBlock (cpu)
   MArrayDataHandle itArray = db.inputArrayValue(TypeII::inputTarget);
   MFnMesh inputTargetMFn(itArray.inputValue().asMesh());
   // c. send meshB vertices from MFnMesh to MAutoCLMem
   // d. enqueue a deform kernel that accesses meshA's & meshB's GPU clmem
}
```

Implementation 1: TypeII forces inputTarget on CPU

```
class PD : public MPxDeformerNode {...};

MStatus PD::initialize(){ ...
   meshBAttr = tAttr.create("meshB", "meshB", MFnData::kMesh); ...
}

// the upstreamNode.worldMesh[0] will be connected to pd.meshB
const MObject PDRegistrationInfo::inputMeshAttribute(){return PD::meshBAttr;}

class PDGPU : public MPxGPUDeformer{ ...
   MAutoCLMem clMemMeshA; // GPU memory storing meshA vertices position
};

DStatus PDGPU::evaluate(MDataBlock& db,.,const MGPUDeformerData& inData,
                                         MGPUDeformerData& outData){
   // a. get meshB vertices from GPU buffer (sPos)
   MGPUDeformerBuffer bufferMeshB = inData.getBuffer(sPositionsName());
   // b. send static meshA to GPU if haven't done before
   if (!clMemMeshA.get()){
      MArrayDataHandle adhMeshA = db.inputArrayValue(DD::input);
      MDataHandle hMeshA = adhMeshA.inputValue();
      MFnMesh meshA(hMeshA.child(DD::inputGeom).data());
      // send meshA vertices from MFnMesh to MAutoCLMem/clMemMeshA
   }
   // c. the downstream deformer/mesh picks up vertices from
   //    sPositionName by default. So we set sPos pointing to meshA
   MGPUDeformerBuffer updatedBufferSPos(sPositionsName(),.,clMemMeshA,.);
   outData.setBuffer(updatedBufferSPos);
   // d. move meshB vertices from sPos to a new buffer called "meshB"
   MUniqueString strMeshB = MUniqueString::intern("meshB");
   MGPUDeformerBuffer updatedBufferMeshB(strMeshB,.,bufferMeshB.buffer(),.);
   outData.setBuffer(updatedBufferMeshB);
   // e. return now and no need to do any mesh deformations
}
```

Implementation 2: PassThrough Deformer (PD)

```
DStatus TypeII::evaluate(MDataBlock& db,.,const MGPUDeformerData& inData,.){
   // a. get meshA vertices from GPU buffer (sPos)
   MGPUDeformerBuffer bufferMeshA = inData.getBuffer(sPositionsName());
   // b. get meshB vertices from GPU buffer
   MUniqueString strMeshB = MUniqueString::intern("meshB");
   MGPUDeformerBuffer bufferMeshB = inData.getBuffer(strMeshB);
   // c. enqueue a deform kernel that accesses meshA's & meshB's GPU clmem
}
```

Implementation 3: TypeII gets meshes from GPU buffers

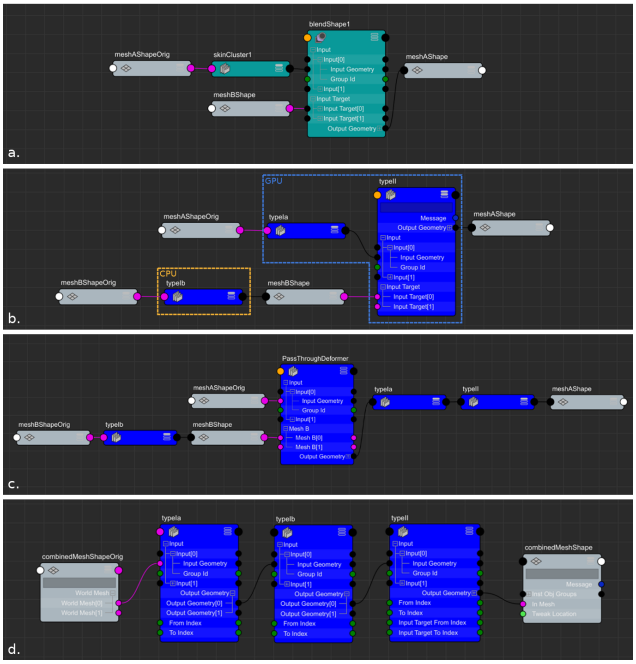

Figure 2: (a) A sample TypeII (blendShape) with a static mesh at inputTarget. (b) A Type II deformer with a non-static mesh at inputTarget (c) All deformers are on GPU with PD. (d) All deformers are on GPU with our combined mesh. The groupParts and tweak nodes are skipped for clarity.