

# 动画大作业文档

张泽宇 2022012117 [zhangzey22@mails.tsinghua.edu.cn](mailto:zhangzey22@mails.tsinghua.edu.cn)

## 1. 运行环境和启动方法

本项目依赖于以下环境：

- Windows 10/11 (在Windows进行测试，硬件系统包含NVIDIA 4070 Laptop显卡加速)
- Python 3.10+ (64-bit)

编译方法：

代码块

```
1 conda activate cv # 创建Python虚拟环境  
2 pip install -r requirements.txt # 安装依赖包
```

额外工具准备：

- 需要使用 `ffmpeg` 包输出视频，因此需要确保环境变量中存在 `ffmpeg`。

关键依赖：

- UI: `PySide6`, `pyvista`, `pyvistaqt`, `vtk`
- 计算: `numpy`, `scipy`, `trimesh`, `networkx`
- GLB: `pygltf`
- 关键帧烘焙: `moderngl`, `PyOpenGL` (legacy/optional)

## 2. 程序模块和逻辑关系

### 2.1 代码结构

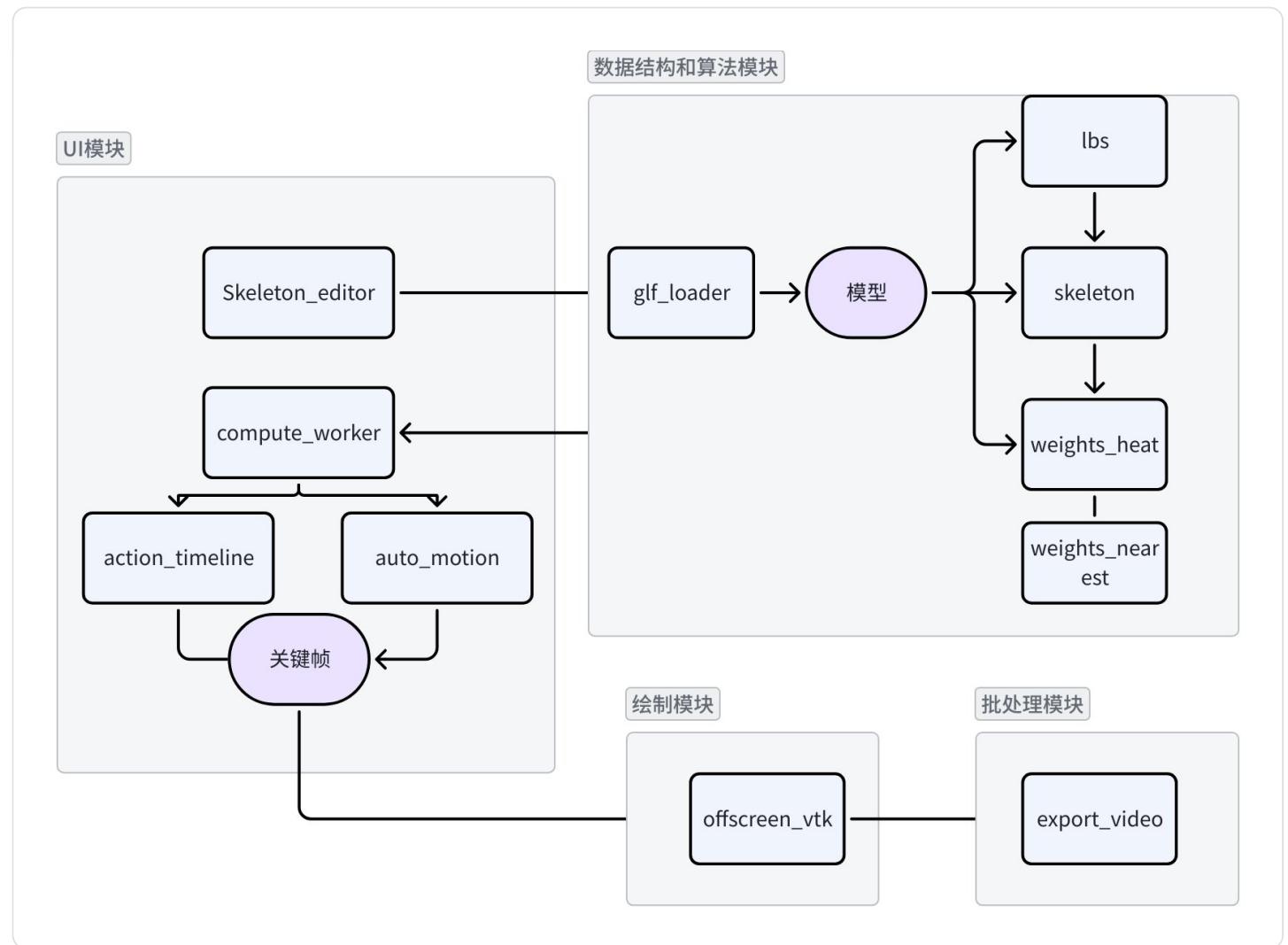
程序中主要包含以下模块：

- `rigging/`: 数据结构和算法模块
  - `mesh_io.py`: (文件I/O、检测和预处理)
  - `skeleton.py` (骨骼相关算法，包含关节、FK、姿势绑定等方法)
  - `weights_heat.py` (热扩散权重算法实现)
  - `weights_nearest.py` (最近邻权重算法实现)

- `lbs.py` (线性蒙皮实现)
  - `gltf_loader.py / gltf_export.py` (GLB文件处理)
  - `rig_io.py` : (NPZ文件处理).
  - `skeleton_optimize.py` : (对于手动绑定的骨骼进行优化)
  - `constraints.py` (IK约束)
- `scene/` :动画模块
    - `timeline.py , actions.py , asset.py , choreography.py` (动画的时间轴, 视频导出等内容)
  - `ui/` :交互模块
    - `skeleton_editor.py` (主UI和流水线处理)
    - `viewport.py` (PyVista/VTK 相关的交互处理)
    - `action_timeline.py` (关键帧/动画相关操作UI)
    - `export_panel.py` (文件相关操作UI)
    - `action_timeline.py` : (在缓存中保存和预览关键帧)
    - `auto_motion.py` : (UI中的示例动作生成器)
  - `render/` :绘制模块
    - `offscreen_vtk.py` (使用VTK绘制png格式的关键帧)
  - `tools/` :批处理模块
    - `bake_frames.py` (批处理烘焙关键帧)
    - `export_video.py` (使用ffmpeg将关键帧拼凑为视频)

## 2.2 数据流图

- 以生成视频为例构建数据流图



视频生成运行的主要流程包括：

1. 在 UI 中加载 glb 格式的文件；
2. （可选）在骨骼编辑模式中编辑骨骼，并选择一种算法重新计算权重；
3. 拉动骨骼存储关键帧
4. 使用vtk渲染关键帧并生成视频

### 3. 功能演示方法

作业中附有功能演示视频

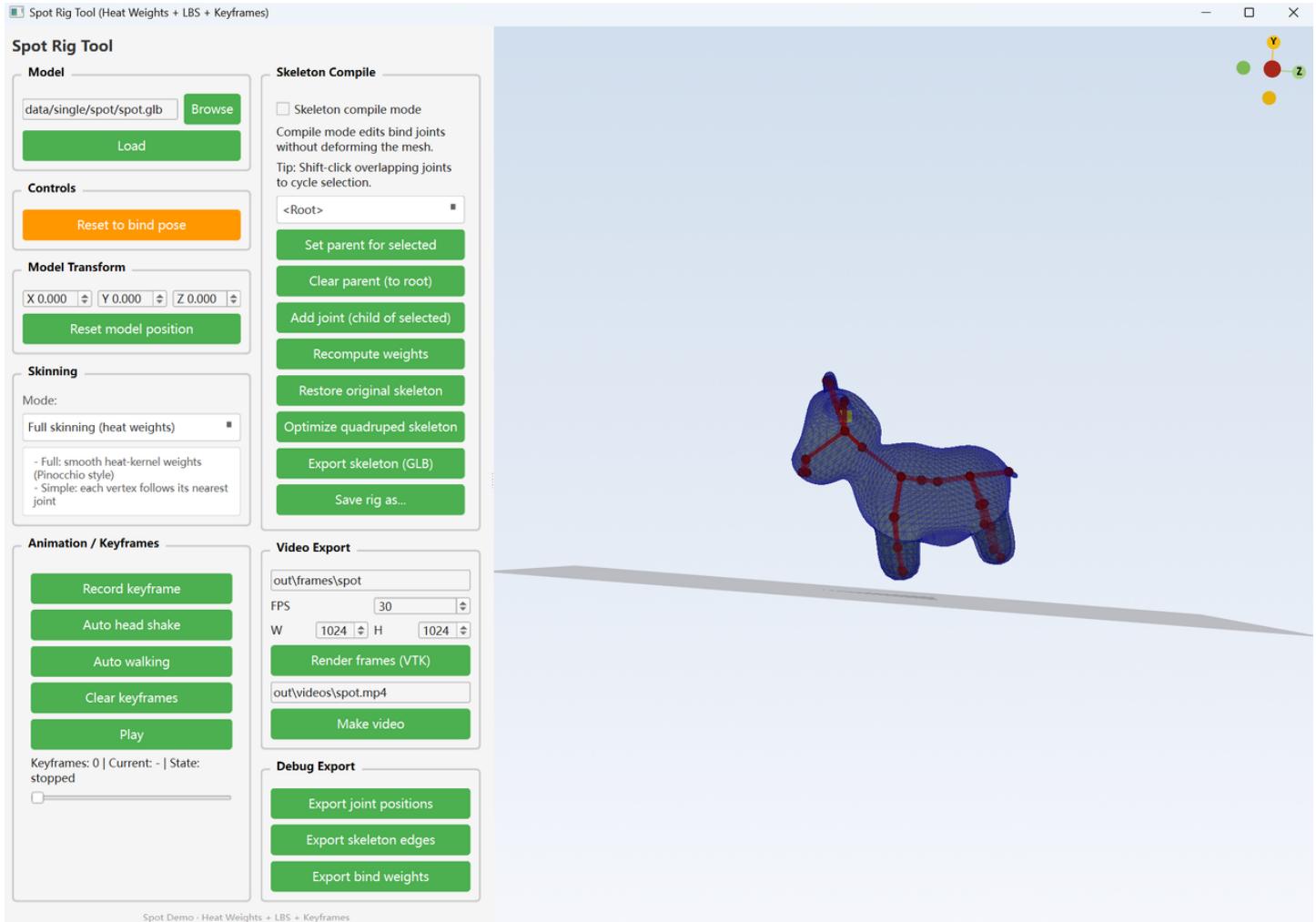
代码块

```
1 python -m u-.ui_simple # UI主入口
```

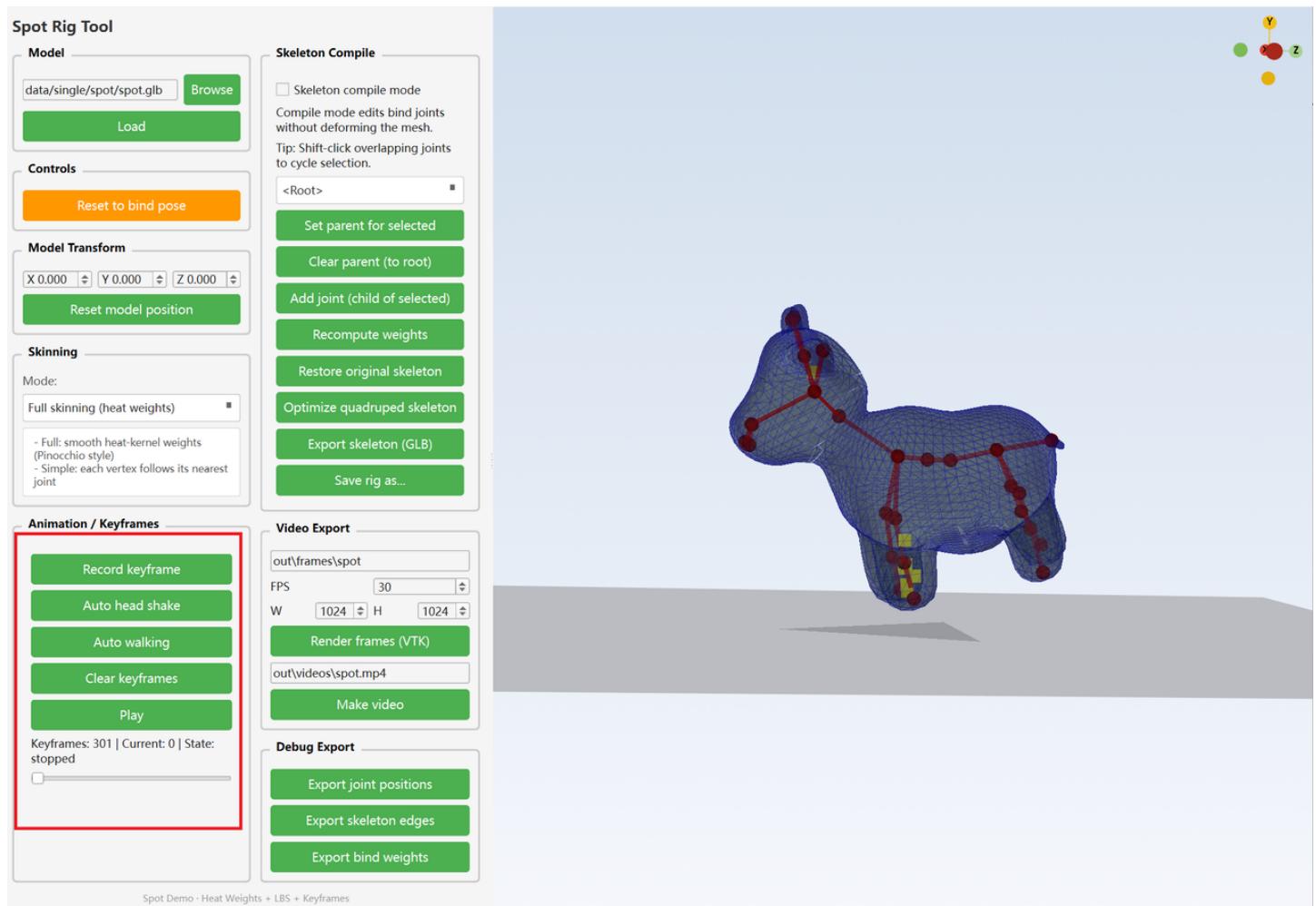
#### 3.1 自动生成视频

- 打开UI，会自动获取文件夹中的 glb 文件并加载
  - 对右上角 Gizmo 鼠标左键可旋转模型角度，滚动鼠标中轮以改变镜头远近

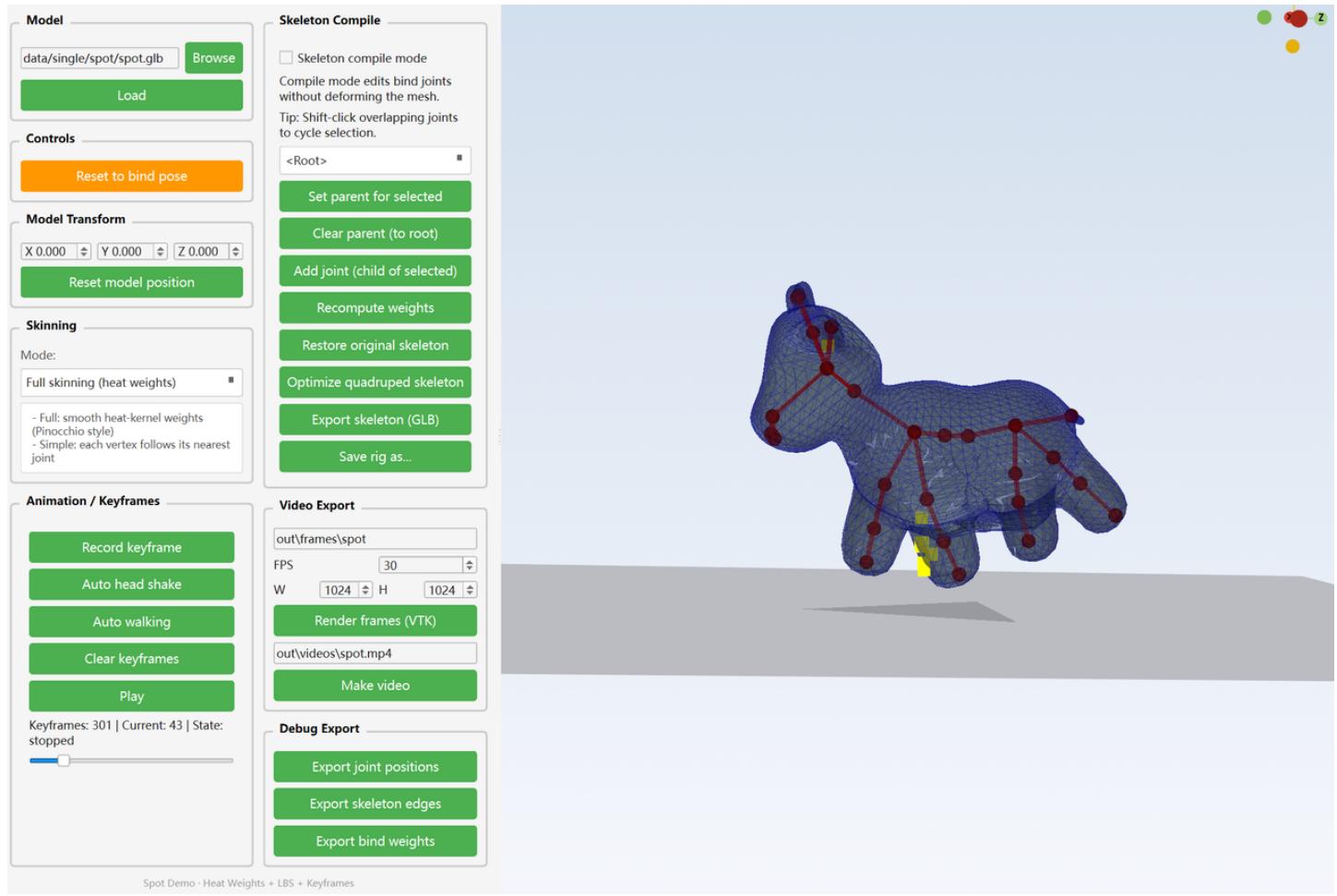
- 右键按住模型拖动可拉动模型，点击 **Reset to bond pose** 可以使模型回复原位置/角度



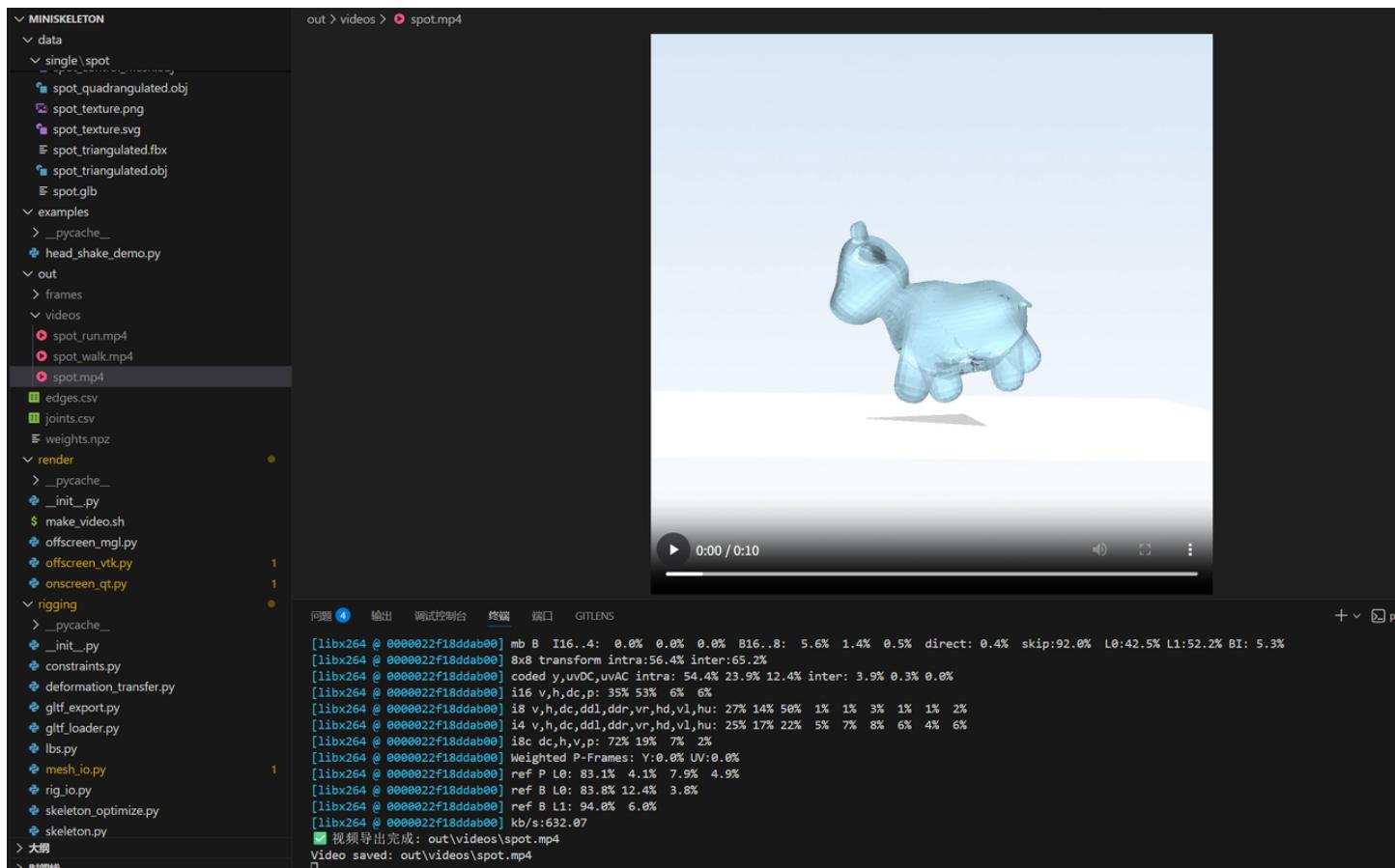
- 点击 **Animation** 中的 **Auto head shake** 或者 **Auto Walking** 会自动生成关键帧
  - 关键帧生成需要一定时间，等到 **keyframe** 出现数字说明生成成功



- 按 Play 或者拉进度条播放动画

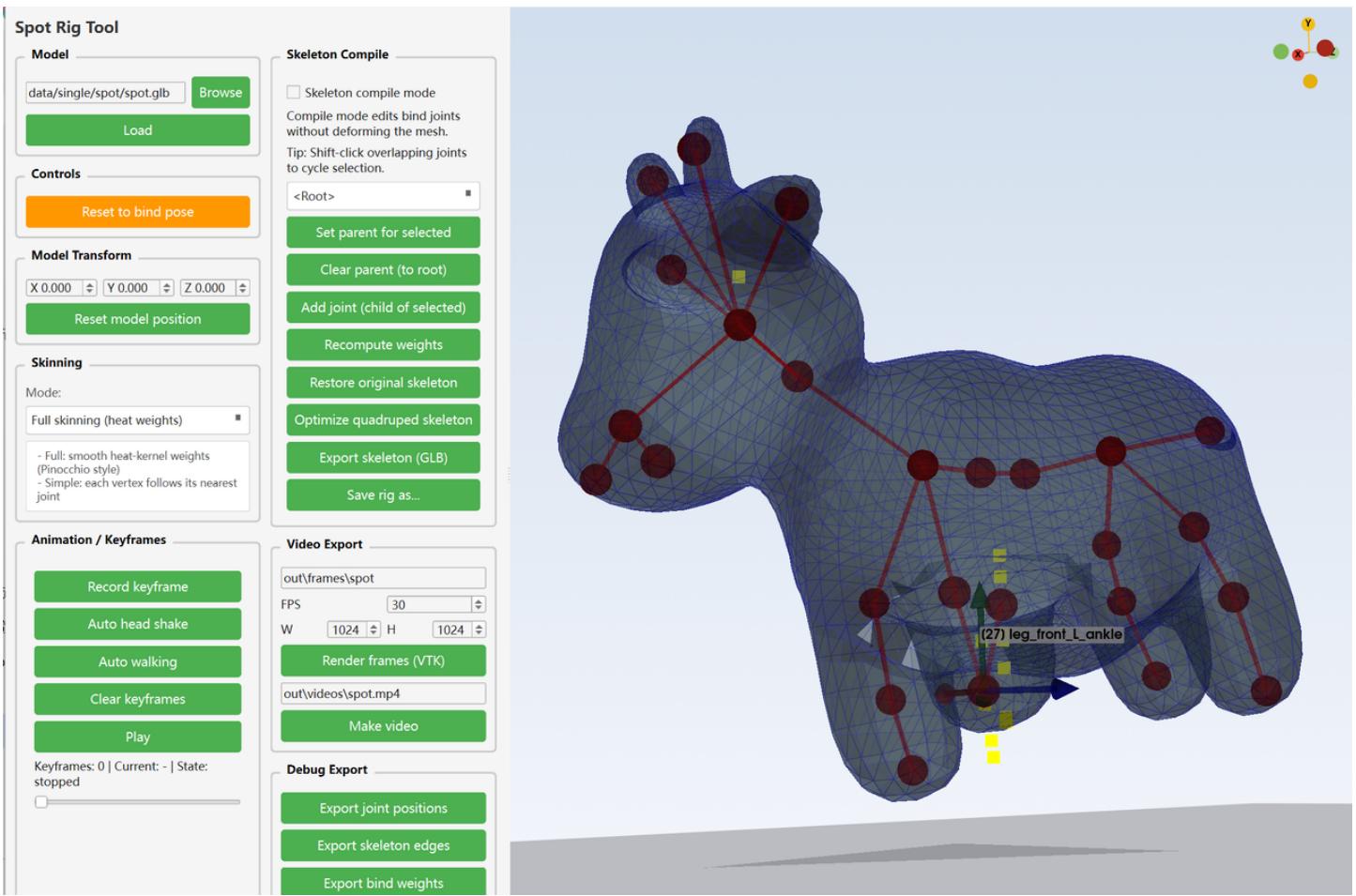


- 按 **Video Export - Make Video** 将视频以 mp4 格式导出

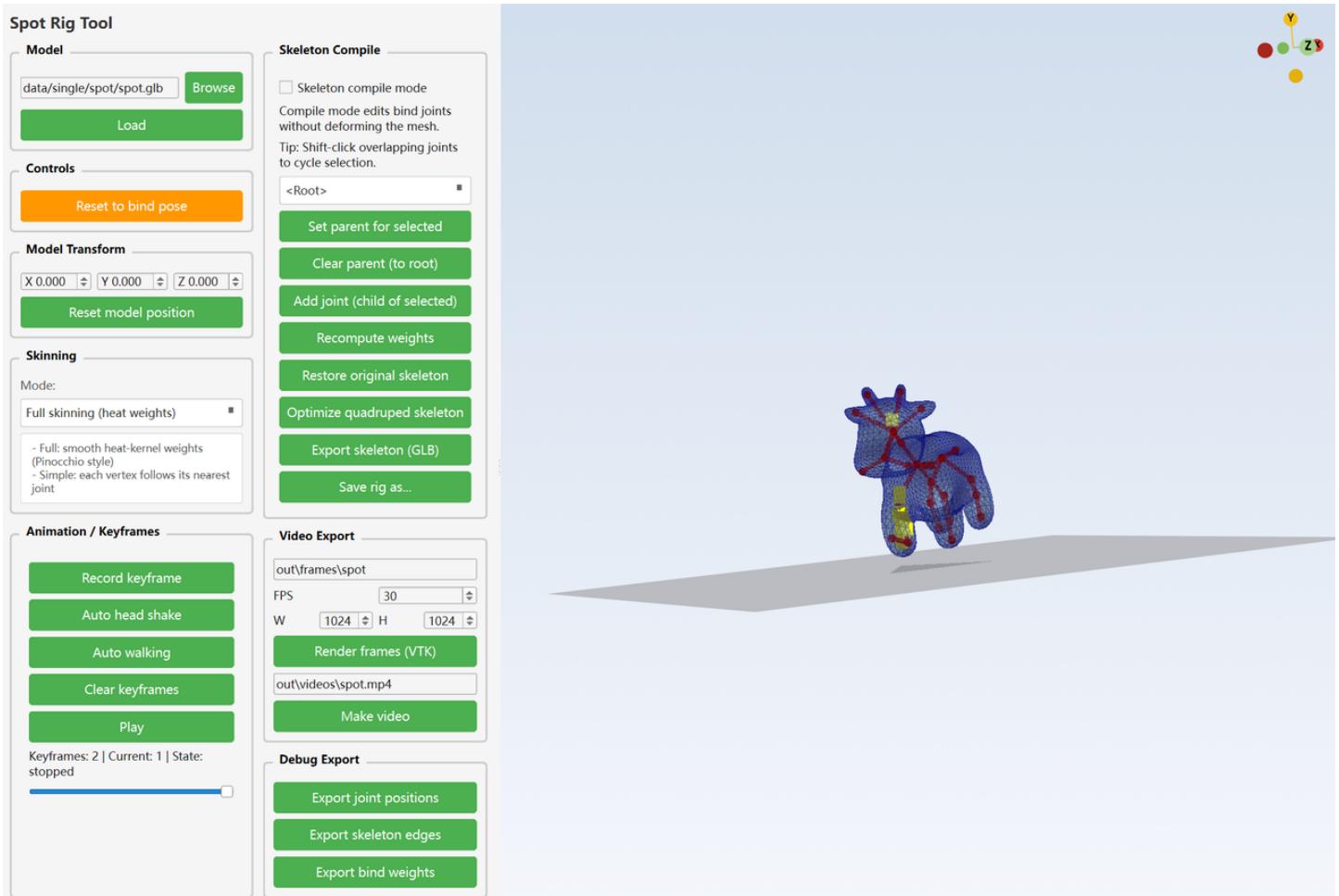


## 3.2 手动生成视频

- 左键双击选中关节，按坐标轴移动关节位置，会自动重算面片（编辑过的关节会有黄色方块标记）
  - 可以在 **Skinning - Mode** 中更改使用的权重计算算法

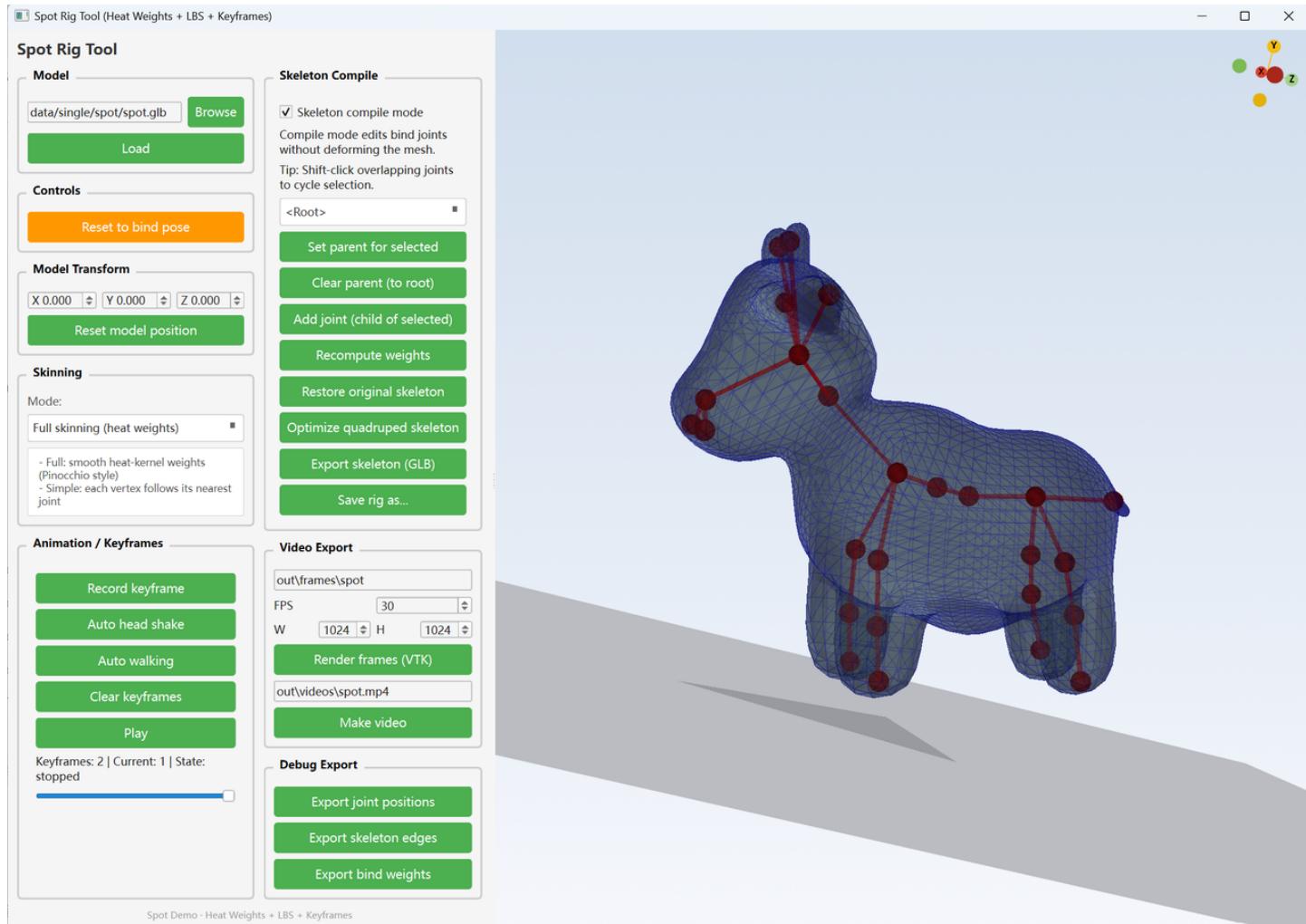


- 点击 Animation - Record Keyframe 将手动动作保存为关键帧，保存关键帧为2个以上是按 Animation - Play 可以生成动画



### 3.3 编辑骨架

- 勾选 `Skeleton Compile - Skeleton compile mode` 进入骨架编辑模式

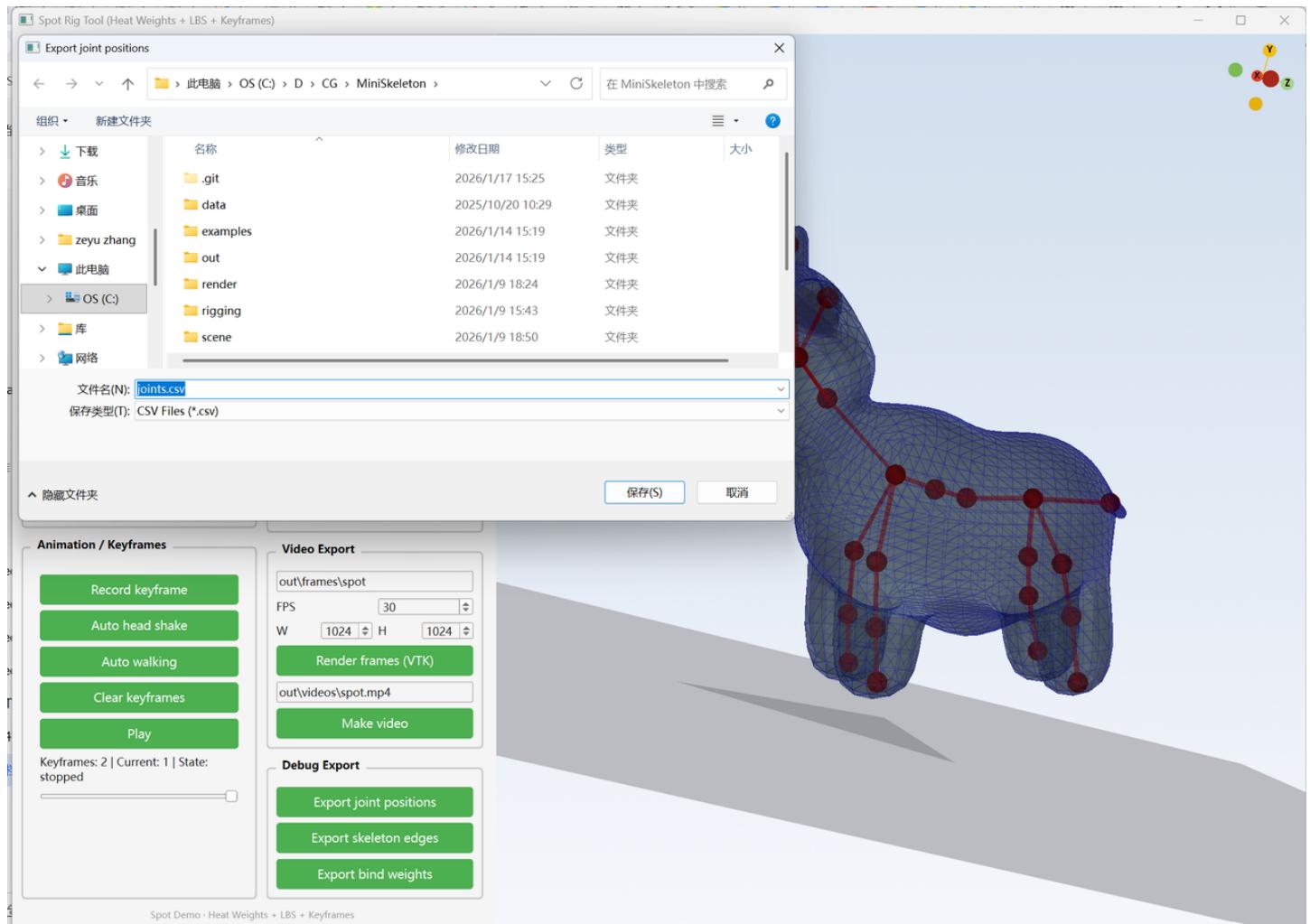


- 骨架编辑模式支持的操作包括

- `Set parent for selected` : 将选中关节的 parent 设置为上方拉选框中的关节
- `Clear parent (to root)` : 清除所选关节的 parent
- `Add joint` : 从选中关节中挤出骨骼
- `Recompute weights` : 重新计算权重
- `Restore original skeleton` : 恢复原有骨骼

### 3.4 导出信息

- 点击 `Debug Export` 中的相关按钮导出对应格式的文件



## 4. 参考文献和引用代码出处

### 4.1 最近骨骼 / 双骨插值

这是一个基础、快速的权重初始化方法，在项目中用作测试，以及 Heat 权重的 warm start 或 fallback。该算法的核心思路如下：

#### 1. 距离度量：

- 把每根骨骼看作线段（父关节 (p)、子关节 (c)）；  
- 对每个顶点 (v)，计算到若干骨骼线段的最短距离 ( $d_{-k}$ )；
- 只保留距离最近的前 ( $K$ ) 根骨骼（比如 2–4 根）。

#### 2. 权重分配：

- 将距离反比转为权重：

$$[\tilde{w}_k = \frac{1}{d_k^\alpha + \epsilon}, \quad w_k = \frac{\tilde{w}_k}{\sum_j \tilde{w}_j}]$$

- 通常 ( $\alpha \in [1, 2]$ )，调整衰减速度。

#### 3. 拓扑一致性优化（双线性/双骨插值）：

- 对于位于肢体中部的顶点 (v)，会同时靠近父骨和子骨；

- 可以基于顶点在骨骼方向上的投影参数 ( $t \in [0, 1]$ ) 做线性插值，调节父子骨权重比例；
- 这样关节弯曲时形变更自然。

双骨差值法仅依赖少量几何操作（点到线段距离），实现简单；但是这种蒙皮方式对骨骼密度和布置较敏感，在复杂拓扑处容易出现权重不连续。

空白 TeX 公式

双骨插值的部分关键代码是：

代码块

```

1  def compute_nearest_bilinear_weights(
2      verts: np.ndarray,                      # (N,3)
3      skel,                                # rigging.skeleton.Skeleton
4      config: Optional[NearestBilinearConfig] = None,
5  ) -> np.ndarray:
6      if config is None:
7          config = NearestBilinearConfig()
8
9      V = np.asarray(verts, dtype=np.float32)
10     N = V.shape[0]
11     parents = skel.parents()
12     edges = _bone_edges_from_parents(parents)    # (B,2)
13     if edges.size == 0:
14         raise ValueError("Skeleton has no bones (no parent-child pairs).")
15
16     # Bind joint positions and bone endpoints
17     J_pos = _global_bind_positions(skel)        # (J,3)
18     A = J_pos[edges[:, 0]]                      # (B,3) parent
19     B = J_pos[edges[:, 1]]                      # (B,3) child
20     AB = B - A
21     seg_lengths = np.linalg.norm(AB, axis=1)    # (B,)
22
23     sigma = config.sigma if (config.sigma is not None) else
24     _auto_sigma(seg_lengths)
25
26     # Prepare output
27     J = J_pos.shape[0]
28     W = np.zeros((N, J), dtype=np.float32)
29
30     # Chunked processing to limit memory footprint
31     bs = int(config.chunk_size)
32     for start in range(0, N, bs):
33         end = min(N, start + bs)
34         Vc = V[start:end]  # (Nv,3)

```

```

34
35     t, d, _ab2 = _project_points_to_segments_batch(Vc, A, B)    # (Nv,B) each
36
37     # pick k nearest bones per vertex
38     K = int(max(1, config.k_bones))
39     if K >= t.shape[1]:
40         K = t.shape[1]
41
42     # indices of K smallest distances along axis=1
43     # argpartition faster than argsort
44     idx_part = np.argpartition(d, K-1, axis=1)[:, :K]           # (Nv,K)
45     # sort those K by distance ascending for stability
46     row_ids = np.arange(idx_part.shape[0])[:, None]
47     d_sorted = np.take_along_axis(d, idx_part, axis=1)
48     order = np.argsort(d_sorted, axis=1)
49     bone_ids = np.take_along_axis(idx_part, order, axis=1)      # (Nv,K)
50     d_sorted = np.take_along_axis(d_sorted, order, axis=1)
51     t_sorted = np.take_along_axis(t, idx_part, axis=1)
52     t_sorted = np.take_along_axis(t_sorted, order, axis=1)
53
54     # radial falloff per bone
55     F = _radial_weight(d_sorted, sigma=sigma, mode=config.falloff,
56     eps=config.eps)   # (Nv,K)
57
58     # accumulate contributions to joints
59     for k in range(K):
60         b_ids = bone_ids[:, k]                                # (Nv,)
61         t_k = t_sorted[:, k]                                 # (Nv,)
62         f_k = F[:, k]                                     # (Nv,)
63
64         j_parent = edges[b_ids, 0]                          # (Nv,)
65         j_child = edges[b_ids, 1]                           # (Nv,)
66
67         w_parent = (1.0 - t_k) * f_k
68         w_child = t_k * f_k
69
70         # scatter-add
71         W[start:end, j_parent] += w_parent
72         W[start:end, j_child]  += w_child
73
74     # prune to max_influences if requested (dense path)
75     if config.max_influences and config.max_influences > 0:
76         mi = int(config.max_influences)
77         # keep top-mi per row
78         # argpartition descending
79         # NOTE: we operate on the chunk only
80         idx_desc = np.argpartition(-W[start:end], kth=mi-1, axis=1)

```

```

80         keep_cols = idx_desc[:, :mi]
81         mask = np.zeros_like(W[start:end], dtype=bool)
82         rr = np.arange(end - start)[:, None]
83         mask[rr, keep_cols] = True
84         W[start:end][~mask] = 0.0
85
86         # renormalize chunk
87         if config.renormalize:
88             rowsum = np.sum(W[start:end], axis=1, keepdims=True)
89             rowsum[rowsum < config.eps] = 1.0
90             W[start:end] /= rowsum
91
92     return W

```

## 4.2 Heat Diffusion 权重

这是主要的权重算法，参考 Pinocchio 等工作中的基于热传导的骨骼权重思想。核心思想是：

把每个骨骼关节当作“热源”，在三角形网格上做热扩散，最终稳态温度场即为该关节的权重分布。

### 4.2.1 离散拉普拉斯与热方程

在网格上，项目构造了离散拉普拉斯矩阵，例如使用 cotangent 权重：

$$[(Lu) * i = \sum *j \in N(i) w_{ij} (u_i - u_j)]$$

对每个关节 (j)，希望求出权重函数，满足离散热方程稳态形式：

$$[(L + \tau I), w^{(j)} = b^{(j)}]$$

### 4.2.2 线性系统与数值求解

对每个关节 (j) 解一个稀疏线性系统：

$$[(L + \tau I), w^{(j)} = b^{(j)}]$$

数值实现上：

- 预先用 `scipy.sparse` 构建好 (`A = L + \tau I`)；
- 对各关节频繁调用 `cg(A, b_j, ...)` (共轭梯度法)；
- 为了避免共轭梯度不收敛，通常会：
  - 选取适度的 ( $\tau$ )；
  - 对 `cg` 设置合理的 `maxiter` 和残差阈值；
  - 失败时 fallback 到最近骨骼权重。

最终把所有关节的结果堆叠，得到权重矩阵，再对每个顶点的权重 `w[i]` 做归一化：

$$[W_{ij} \leftarrow \frac{\max(W_{ij}, 0)}{\sum_k \max(W_{ik}, 0) + \varepsilon}]$$

## 2.2.3 与 LBS 的结合

一旦有了 ( $W$ ) 和 FK 算出的关节矩阵 ( $T_j$ )，就可以进行标准的**线性混合蒙皮**：

$$[v'_i = \sum_j W_{ij}; (T_j, T_j^{\text{bind}^{-1}}); \hat{v}_i]$$

其中：

- ( $\hat{v}_i$ ) 为 Bind Pose 下的齐次坐标；
- ( $T_j^{\text{bind}}$ ) 由 GLB 的 `inverseBindMatrices` 或 Skeleton 的绑定位姿推得。

Heat 权重的好处包括以下要点：

- 权重场在模型表面**连续且平滑**；
- 在关节附近自然实现多骨“混合”控制，弯曲时不易产生明显折痕；
- 对骨架布置有一定鲁棒性，不要求手动刷权。

Heat权重的关键算法内容包括：

代码块

```

1      # σ 自动估计：中位骨段长度的一半（经验值）
2      if cfg.sigma is None:
3          med = float(np.median(seg_len))
4          cfg_sigma = max(0.5 * med, 1e-3)
5      else:
6          cfg_sigma = float(cfg.sigma)
7
8      # 计算网格邻接与拉普拉斯
9      neighbors, L = compute_vertex_adjacency(mesh)
10     if L is None:
11         raise RuntimeError("No Laplacian available (SciPy missing?).")
12
13     # 随机游走标准化 Laplacian: L_rw = D^{-1} L
14     deg = np.asarray(L.sum(axis=1)).ravel()
15     deg[deg < 1e-16] = 1.0
16     Dinv = sp.diags(1.0 / deg, 0, shape=L.shape)
17     L_rw = Dinv @ L
18
19     # A = I - tau * L_rw
20     I = sp.identity(L.shape[0], format="csr", dtype=np.float32)
21     A_sys = (I - cfg.tau * L_rw).tocsr()
22
23     # 预处理/分解（一次，多 RHS 复用）
24     solver_mode = cfg.solver.lower()
```

```

25     do_splu = (solver_mode == "splu" or solver_mode == "auto")
26     lu = None
27     if do_splu:
28         try:
29             lu = spla.splu(A_sys.tocsc())
30         except Exception:
31             # 回退到 CG
32             lu = None
33             solver_mode = "cg"

```

## 4.3 参考文献

名称	文献链接
Ilya Baran and Jovan Popović. 2007. Automatic rigging and animation of 3D characters. ACM Trans. Graph. 26, 3 (July 2007), 72–es.	<a href="https://www.cs.toronto.edu/~jacobson/seminar/baran-and-popovic-2007.pdf">https://www.cs.toronto.edu/~jacobson/seminar/baran-and-popovic-2007.pdf</a>
Automatic Skinning using the Mixed Finite Element Method (Arxiv)	<a href="https://arxiv.org/html/2408.04066v1">https://arxiv.org/html/2408.04066v1</a>

## 4.4 学术诚信声明

本项目使用了 GPT-5.2-Codex 生成部分代码（30%），所有代码均经过人工审阅，并且未使用任何其他人撰写的代码。本项目所有提交和更改记录均可以在<https://github.com/Alchuang22-dev/MiniSkeleton>访问。