GPT4Motion: Scripting Physical Motions in Text-to-Video Generation via Blender-Oriented GPT Planning

Jiaxi Lv[1,2*] Yi Huang[1,2*] Mingfu Yan[1,2*] Jiancheng Huang[1,2] Jianzhuang Liu[1]
Yifan Liu[1] Yafei Wen[3] Xiaoxin Chen[3] Shifeng Chen[1†]
[1]Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences,
[2]University of Chinese Academy of Sciences, [3]VIVO AI Lab
{jx.lv1, yi.huang, mf.yan, jc.huang, jz.liu, yf.liu2, shifeng.chen}@siat.ac.cn
{yafei.wen, xiaoxin.chen}@vivo.com

Supplementary Material

GPT4Motion AnimateDiff [1] ModelScope [8] Text2Video-Zero [4] DirecT2V [2]

Figure 1. Comparison of the video results generated by different text-to-video models under different physical conditions. *Best viewed with Acrobat Reader for animation.*

In this supplement material, we further compare the generation ability of GPT4Motion and four baselines for different physical conditions in Sec. 1. In Sec. 2, we introduce the quantitative metrics we used. In Sec. 3, we describe the details of our settings for Blender. In Sec. 4 and Sec. 5, we show how GPT-4 generates encapsulated Python functions via prompts, and how GPT-4 incorporates its own knowledge of physics to control the motion of objects, respectively.

1. More Comparison with Baselines

In the main paper, we have compared GPT4Motion with four baselines (AnimateDiff [1], ModelScope [8], Text2Video-Zero [4], and DirecT2V [2]) on three scenarios (rigid object drop and collision, cloth draping and swinging, and liquid flow). Here, we further conduct an experiment on dynamic effects of a T-shirt being blown by the wind under three wind strengths. The results are shown in Figure 1, where the seed is randomly chosen and fixed in all the generations. We can see that these baselines all fail to generate videos that match the user prompts and are unable to control the intensity of physical phenomena solely based on the linguistic descriptions. In contrast, our GPT4Motion not only precisely designs the parameters of Blender encapsulated functions (such as wind strength) through GPT-4, but also leverages Blender's physics engine to simulate the complex flapping and twisting dynamics of the T-shirt in the wind.

2. Quantitative Evaluation Metrics

Here, we introduce the metrics employed in the main paper:

1. *Motion Smoothness* [3]. This metric evaluates the smoothness of motion in generated videos, ensuring it conforms to the physical laws of the real world. The evaluation utilizes motion priors from the video frame interpolation model [5] to assess the smoothness of generated motions.
2. *Temporal Flickering* [3]. This metric identifies imperfections in temporal consistency within generated videos, especially in local and high-frequency details. The method involves analyzing static frames and computing the mean absolute difference across frames. These values are then subtracted from 255 and normalized between 0 and 1.
3. *CLIP-Score* [6]. This metric is designed to assess Text-Video Consistency. It leverages a pretrained ViT-B/32 CLIP model [7] as a feature extractor to calculate the cosine similarity between each video frame and the associated text.

3. Blender Settings

We use Blender to generate two sequences of scene depth maps and edge maps. The edge maps are obtained by Blender's built-in Freestyle[1] feature, which is an advanced rendering technique for non-photorealistic line drawings from 3D scenes. It not only allows for various line styles, such as hand-drawn or sketch, but also serves as an independent rendering pass without losing edge information of the scene and additional post-processing. Moreover, we normalize the scene's Z-channel and render it as an independent depth map. Since the rendering of edge maps and depth maps is unrelated to ray tracing, we choose Blender's "Workbench" rendering mode. This mode avoids the substantial computational resources required for ray tracing, significantly enhancing rendering efficiency. In our experiments, completing the entire rendering process using Blender scripts takes only 2 to 3 minutes, with most of the time spent on calculating physical effects.

4. GPT-4's Generation of Encapsulated Python Functions for Blender

Blender incorporates an independent Python library, enabling the transformation of various manual processes within the software to automate tasks through its Python API. Leveraging GPT-4's capabilities of grasping Blender's Python API, we can rapidly generate encapsulated Python functions for Blender. This involves inputting specific functional goals and the associated Blender operation steps into GPT-4. Consequently, GPT-4 can autonomously create Python code that corresponds to these steps, effectively encapsulating them into Python functions. One example is given below.

User:

```
You are an expert in Blender
and Python. Please generate
encapsulated Blender Python
functions by following our
step-by-step instructions,
writing the Blender Python API
corresponding to each step,
designing each function's
name, and explaining its
functionality and the meaning of
each parameter in the docstring.
```

GPT-4:

```
Sure, I'd be happy to help
with that. Please go ahead and
provide me with the step-by-step
instructions for the Blender
Python functions you need, and
I will write the corresponding
code for you.
```

[1] https://docs.blender.org/manual/en/latest/render/freestyle/introduction.html

User:

Please build a function to remove all objects, cameras and lights from the initial scene.

GPT-4:

Certainly! To create a Blender Python function that removes all objects, cameras, and lights from the initial scene, you can define a function like this:

```python
# bpy is the Python module used for scripting within Blender
import bpy

def clear_scene():
    """
    Clears all objects from the current Blender scene.

    This function selects all objects in the scene and deletes them.
    It is useful when starting a new scene setup or resetting the scene to a blank state.

    No inputs or outputs.

    Typically called at the beginning of a script when starting a new scene setup.
    """
    bpy.ops.object.select_all(action='SELECT')
    bpy.ops.object.delete()
```

You can call this function whenever you need to clear the scene of all types of objects. Just make sure to be in the correct context where the scene is accessible.

User:

Please now create two physics functions, one that adds collision properties to the inputted object, and another that adds rigid body properties to the object. Then, construct a function to create a floor that scales it up 50 times in all directions, with physics set to collision and rigid body (passive, with elasticity set to 1).

GPT-4:

To accomplish this task, we'll create three functions in Blender's Python API:

1. Add Collision Properties: This function will add collision properties to a specified object.
2. Add Rigid Body Properties: This function will add rigid body properties to a specified object.
3. Create a Floor with Physics: This function will create a floor, scale it, and set its physics properties to collision and rigid body.

```python
import bpy

def add_collision(obj):
    """
    Adds a collision modifier to a Blender object if it does not already have one.

    Parameters:
    - obj (Blender Object): The object to which the collision modifier will be added.

    Returns:
    - None, but the object will have a collision modifier added to it if it wasn't present before.

    This function is used in physics simulations where it's necessary for objects to interact with each other,
    such as rigid body or soft body simulations. The collision modifier makes the object a collider in the physics
    simulation, allowing other
```

```
        objects to bounce off or slide along its surface.
    """
    if "Collision" not in obj.modifiers:
        bpy.ops.object.modifier_add(type='COLLISION')

def add_rigid_body(obj, mass=1, elasticity=0.5, rigid_body_type='ACTIVE'):
    """
    Adds a rigid body physics characteristic to a Blender object.

    Args:
    - obj (Blender Object): The object to which the rigid body physics will be applied.
    - mass (float): The mass of the object, default is 1.
    - elasticity (float): The restitution (bounciness) of the object, default is 0.5.
    - rigid_body_type (str): The type of rigid body, can be 'ACTIVE' or 'PASSIVE', default is 'ACTIVE'.

    Returns:
    - None, but the object is now a rigid body with the specified physics properties.

    This function is crucial for physics simulations, defining how the object behaves under physical forces.
    """
    bpy.ops.object.select_all(action='DESELECT')
    obj.select_set(True)
    bpy.context.view_layer.objects.active = obj
    bpy.ops.rigidbody.object_add()
    obj.rigid_body.type = rigid_body_type
    obj.rigid_body.mass = mass
    obj.rigid_body.restitution = elasticity
    obj.rigid_body.collision_shape = 'MESH'

def create_floor(elasticity=1):
    """
    Creates a floor plane in Blender, scales it, and sets it up with collision and rigid body physics.
    The created floor is scaled to be large enough to act as a ground plane for most scenes.

    Parameters:
    - elasticity (float): The restitution or 'bounciness' of the floor. A value of 1 means perfectly elastic, while 0 means no elasticity. Default is 1.
    """
    bpy.ops.mesh.primitive_plane_add(size=1, enter_editmode=False, align='WORLD', location=(0, 0, 0))
    floor = bpy.context.active_object
    floor.scale = (50, 50, 50)
    floor.name = 'GROUND'
    add_collision(floor)
    add_rigid_body(floor, rigid_body_type='PASSIVE', elasticity=elasticity)
```

These functions together allow you to dynamically add physics properties to objects and create a large, physical floor in your Blender scene. Remember to run these functions in a Blender Python scripting context where the bpy module is accessible.

After generating the function, we enhance its usability for GPT-4 by incorporating instructions (such as the strength interval of the wind) in the function's docstring. This ensures that GPT-4's usage aligns more closely with our expectations.

5. Combination of Blender and GPT-4's Physics Knowledge

We discover that GPT4Motion can not only use GPT-4's semantic understanding and coding capabilities to generate Blender Python scripts, but it can also leverage GPT-4's physical knowledge to control object motion. For example, when prompted with "a basketball thrown towards the camera.", we find that GPT-4 can calculate the time it takes for the basketball to fall, based on the height difference between the initial position of the basketball and the camera. This calculation helps to determine the initial velocity required for the basketball to fly to the camera. The speed calculation code generated by GPT-4 is as follows:

```
# Setup the basketball
basketball = setup_object(
    os.path.join(ASSETS_PATH, "basketball.obj"),
    "Basketball",
    size=0.24,  # Standard basketball diameter in meters
    mass=0.625,  # Standard basketball mass in kilograms
    position=(0, 0, 4)  # Initial position for the basketball
)

# Calculate the initial velocity for the basketball to hit the camera
g = 9.81  # Acceleration due to gravity in m/s^2
camera_height = 1.8521
height_difference = 4 - camera_height
time_to_fall = math.sqrt(2 * height_difference / g)
initial_horizontal_velocity = 13.665 / time_to_fall  # Distance to camera along Y-axis

# Set the initial velocity and rotation for the basketball
add_initial_velocity_for_rigid_body(
    basketball,
    initial_velocity=(0, initial_horizontal_velocity, 0),
    initial_rotation=(0, 0, 0)  # No initial rotation
)
```

In the above script, GPT-4 first calculates the height difference between the initial position of the basketball and the camera, which determines the time required for the basketball to fall to the same height as the camera. Subsequently, GPT-4 calculates the distance between the basketball and the camera along the Y-axis to determine the required initial velocity of the basketball. This process effectively integrates basic principles of physics, such as the equations of motion, to solve a practical problem in a simulated environment like Blender.

References

[1] Yuwei Guo, Ceyuan Yang, Anyi Rao, Yaohui Wang, Yu Qiao, Dahua Lin, and Bo Dai. Animatediff: Animate your personalized text-to-image diffusion models without specific tuning. *arXiv preprint arXiv:2307.04725*, 2023. 1, 2

[2] Susung Hong, Junyoung Seo, Sunghwan Hong, Heeseong Shin, and Seungryong Kim. Large language models are frame-level directors for zero-shot text-to-video generation. *arXiv preprint arXiv:2305.14330*, 2023. 1, 2

[3] Ziqi Huang, Yinan He, Jiashuo Yu, Fan Zhang, Chenyang Si, Yuming Jiang, Yuanhan Zhang, Tianxing Wu, Qingyang Jin, Nattapol Chanpaisit, Yaohui Wang, Xinyuan Chen, Limin Wang, Dahua Lin, Yu Qiao, and Ziwei Liu. VBench: Comprehensive benchmark suite for video generative models. *arXiv preprint arXiv:2311.17982*, 2023. 2

[4] Levon Khachatryan, Andranik Movsisyan, Vahram Tadevosyan, Roberto Henschel, Zhangyang Wang, Shant Navasardyan, and Humphrey Shi. Text2video-zero: Text-to-image diffusion models are zero-shot video generators. In *ICCV*, 2023. 1, 2

[5] Zhen Li, Zuo-Liang Zhu, Ling-Hao Han, Qibin Hou, Chun-Le Guo, and Ming-Ming Cheng. Amt: All-pairs multi-field transforms for efficient frame interpolation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9801–9810, 2023. 2

[6] Yaofang Liu, Xiaodong Cun, Xuebo Liu, Xintao Wang, Yong Zhang, Haoxin Chen, Yang Liu, Tieyong Zeng, Raymond Chan, and Ying Shan. Evalcrafter: Benchmarking and evaluating large video generation models. 2023. 2

[7] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pages 8748–8763, 2021. 2

[8] Jiuniu Wang, Hangjie Yuan, Dayou Chen, Yingya Zhang, Xiang Wang, and Shiwei Zhang. Modelscope text-to-video technical report. *arXiv preprint arXiv:2308.06571*, 2023. 1, 2