



Day 7 : 製造模擬 Case Study - 下

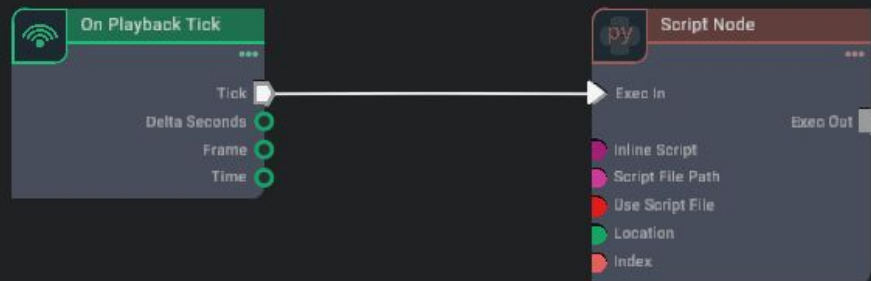
...

2024/07/30

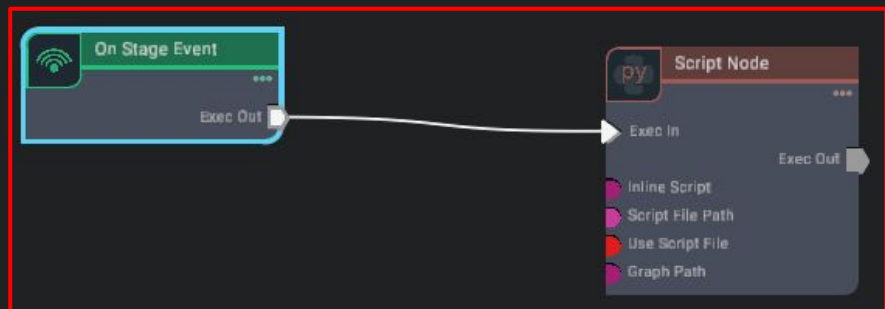
當按下停止模擬，刪除所有創建的貨物(1/6)

如果要刪除在模擬運行中不斷生成的箱子，可以修改 Action Graph，偵測到使用者按下停止模擬的按鈕，就執行刪除貨物的程式碼。

action_graph_0



create_box.py



delete_box.py

當按下模擬停止，刪除所有創建的貨物(2/6)

extension.p
y

所以要去修改生成 Action Graph 的區塊，
 多加上 `on_stage_event(OnStageEvent)` 及
`delete_box_node(ScriptNode)`，
 並為 `delete_box_node` 加上一個 inputs 欄位 `graph_path`，
 為了取得貨物名稱會需要用到 `graph_path`。

```
keys.CREATE_NODES: [
    ("on_playback_tick", "omni.graph.action.OnPlaybackTick"),
    ("create_box_node", "omni.graph.scriptnode.ScriptNode"),
    ("on_stage_event", "omni.graph.action.OnStageEvent"),
    ("delete_box_node", "omni.graph.scriptnode.ScriptNode")
],

keys.CREATE_ATTRIBUTES: [
    ("create_box_node.inputs:location", "pointd[3]"),
    ("create_box_node.inputs:index", "int"),
    ("delete_box_node.inputs:graph_path", "string")
],
```

當按下模擬停止，刪除所有創建的貨物(3/6)

extension.py

接著來設定偵測的 stage event 是Simulation Stop Play,
graph_path 這邊會設定 Action Graph 的路徑給它。

```
graph_path = f"/action_graph_{count}"

keys.SET_VALUES: [
    ("create_box_node.inputs:usePath", True),
    ("create_box_node.inputs:scriptPath", os.path.join(extension_data_path, 'create_box.py')),
    ("create_box_node.inputs:location", (x, y, z)),
    ("create_box_node.inputs:index", count),
    ("on_stage_event.inputs:eventName", "Simulation Stop Play"),
    ("delete_box_node.inputs:usePath", True),
    ("delete_box_node.inputs:scriptPath", os.path.join(extension_data_path, 'delete_box.py')),
    ("delete_box_node.inputs:graph_path", graph_path)
],
keys.CONNECT: [
    ("on_playback_tick.outputs:tick", "create_box_node.inputs:execIn"),
    ("on_stage_event.outputs:execOut", "delete_box_node.inputs:execIn")
],
```

當按下模擬停止，刪除所有創建的貨物(4/6)

extension.py

生成貨物的程式碼這邊，
多加上一個 `box_path_list` 來記錄所有生成的貨物的路徑。

```
31 def compute(db: og.Database):
32     state = db.per_instance_state
33     index = db.inputs.index
34
35     if time.time() - state.last_time > 5:
36         prim_path = f'/World/cardbox_{index}_{str(state.count).zfill(2)}'
37         state.box_path_list.append(prim_path)
38
39         omni.kit.commands.execute('CreatePayload',
40                                   usd_context=omni.usd.get_context(),
41                                   path_to=prim_path,
42                                   asset_path=os.path.join(state.extension_data_path, 'cardbox.usd'),
43                                   instanceable=False)
```

當按下模擬停止，刪除所有創建的貨物(5/6)

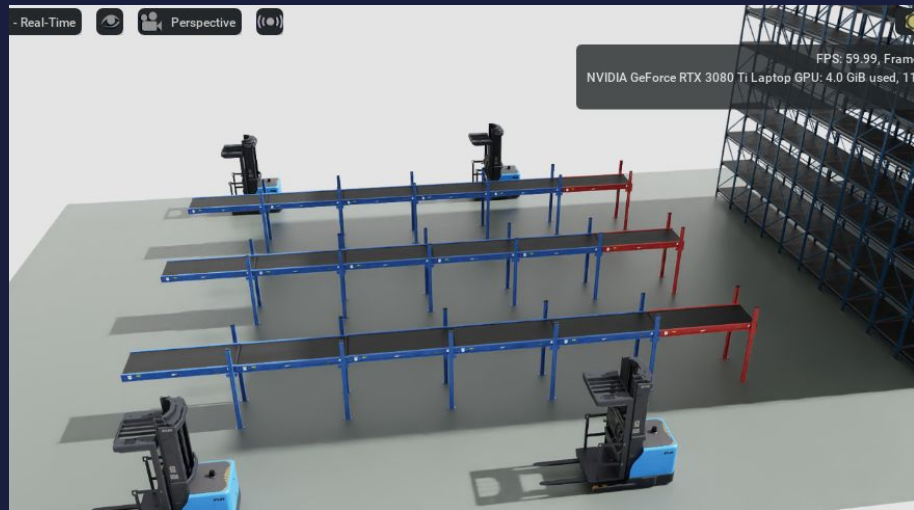
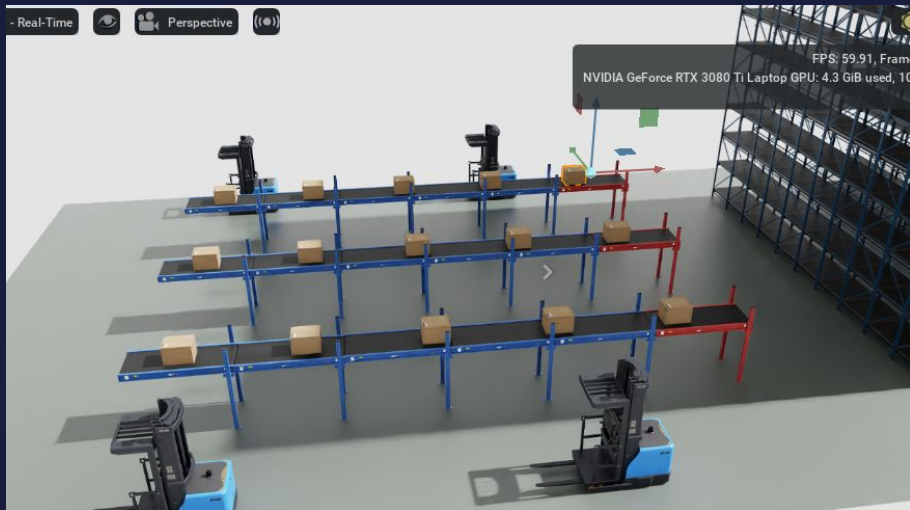
delete_box.py

最後，在新創建的刪除貨物的程式碼中，
先取得生成貨物程式碼的 `state`，就可以拿到它的
`box_path_list`，
根據這個 `list` 內紀錄的路徑，就可以把所有貨物刪掉了。

```
26 def compute(db: og.Database):
27     node = og.get_node_by_path(f"{db.inputs.graph_path}/create_box_node")
28     state = db.per_instance_internal_state(node)
29     state.count = 1
30
31     for box_path in state.box_path_list:
32         omni.kit.commands.execute('DeletePrims',
33             paths=[Sdf.Path(box_path)],
34             destructive=False)
35
36     return True
```


當按下模擬停止，刪除所有創建的貨物(6/6)

測試一下模擬開始後生成的貨物，
是不是在按下模擬停止之後都被刪除
了。



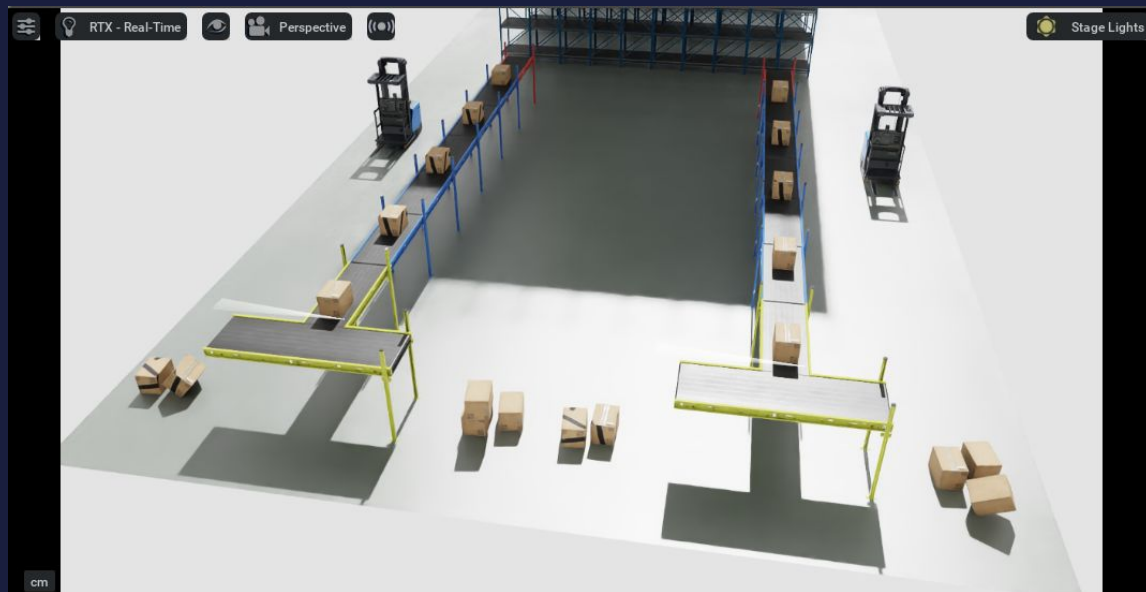
練習時間一

1. 將上次課程的 `create-box` 插件，
試著修改成按下模擬停止後，就會把所有創建的貨物刪掉。

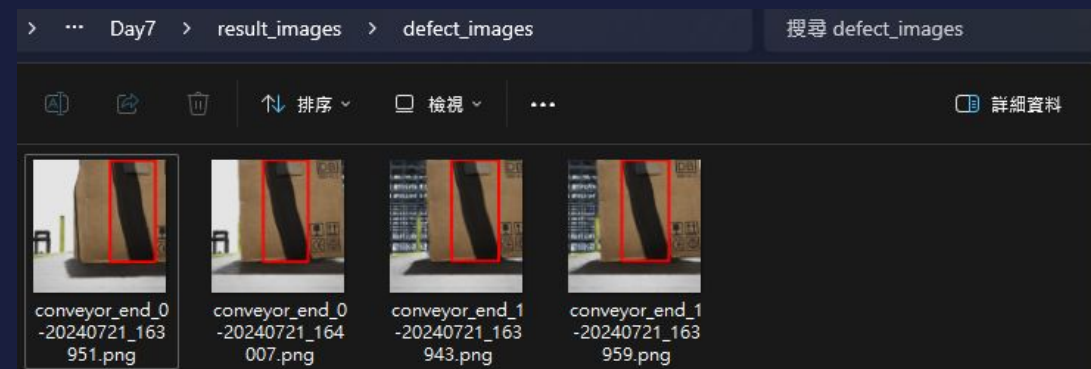
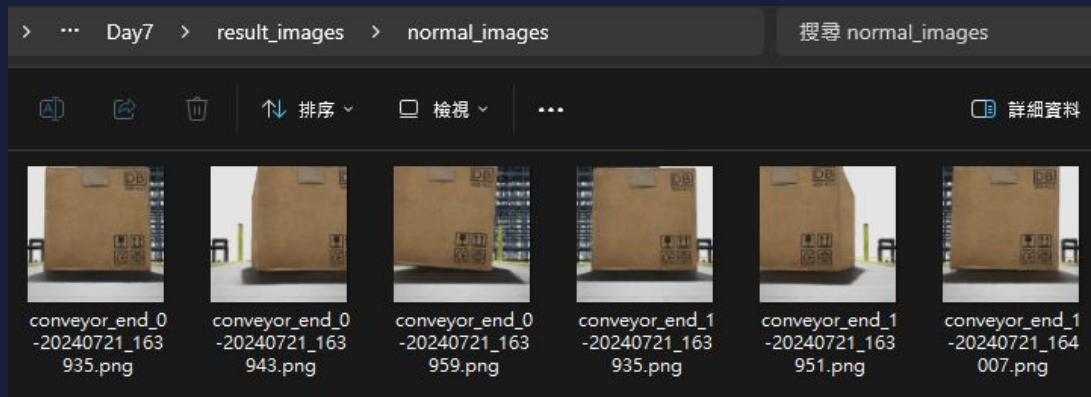
(不一定要用前面提到的做法，
比方你也可以直接掃描整個場景中所有物件的路徑，
把路徑包含 `cardbox` 的物件都刪掉。)

模擬瑕疵檢測系統(1/21)

再來試著模擬一個瑕疵檢測系統，我們希望在輸送帶的尾端，加上一個T字型的輸送帶，當貨物運到 T字型輸送帶，就會用影像辨識的方式確認貨物是否有瑕疵，根據是否有瑕疵會輸送到不同的方向。



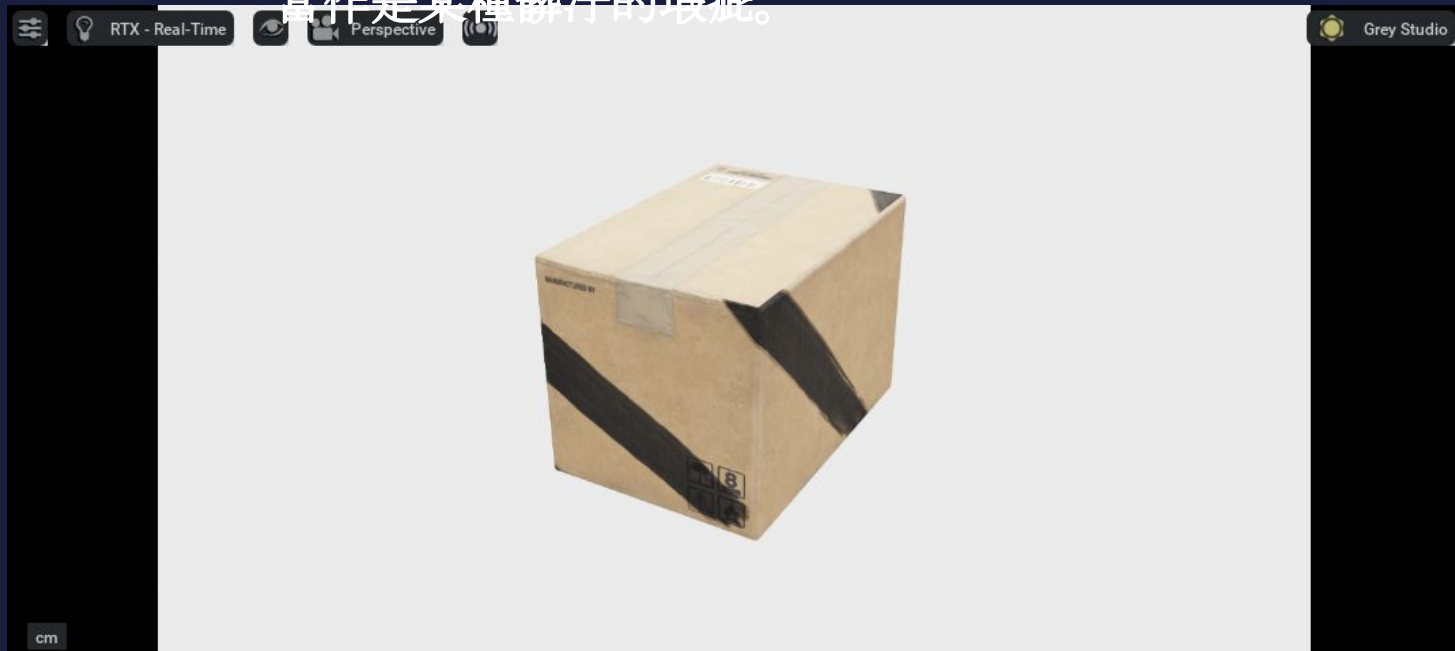
模擬瑕疵檢測系統(2/21)



另外，根據是否有瑕疵，
會將檢測圖片
儲存於不同的資料夾，
供後續確認檢測效果用。

模擬瑕疵檢測系統(3/21)

首先，需要準備瑕疵的貨物，
這邊把原本貨物的顏色貼圖畫上一些黑線
'
當作是某種髒汙的瑕疵。



模擬瑕疵檢測系統(4/21)

create_box.py

在生成貨物程式碼的 `setup` 這邊，
多加上 `asset_path_list`，用來記錄要生成的貨物種類。

```
18 def setup(db: og.Database):  
19     state = db.per_instance_state  
20     state.box_path_list = []  
21     state.asset_path_list = ['cardbox.usd', 'defect_cardbox.usd']  
22     state.count = 1  
23     state.last_time = time.time()
```

模擬瑕疵檢測系統(5/21)

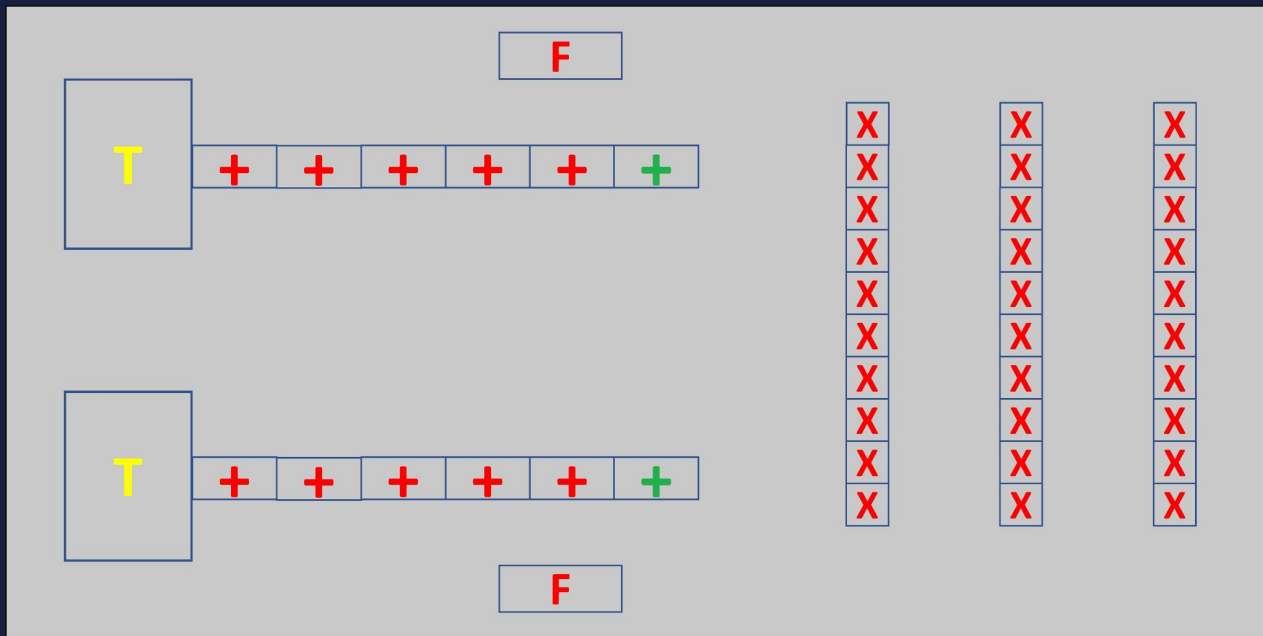
create_box.py

compute 這邊做紅框內的改寫，
這樣每次生成貨物，就會從準備的貨物 USD 檔中隨機挑一個生成。

```
32 def compute(db: og.Database):
33     state = db.per_instance_state
34     index = db.inputs.index
35
36     if time.time() - state.last_time > 8:
37         prim_path = f'/World/cardbox_{index}_{str(state.count).zfill(2)}'
38         state.box_path_list.append(prim_path)
39
40         asset_path = os.path.join(state.extension_data_path, random.choice(state.asset_path_list))
41         omni.kit.commands.execute('CreatePayload',
42             usd_context=omni.usd.get_context(),
43             path_to=prim_path,
44             asset_path=asset_path,
45             instanceable=False)
```

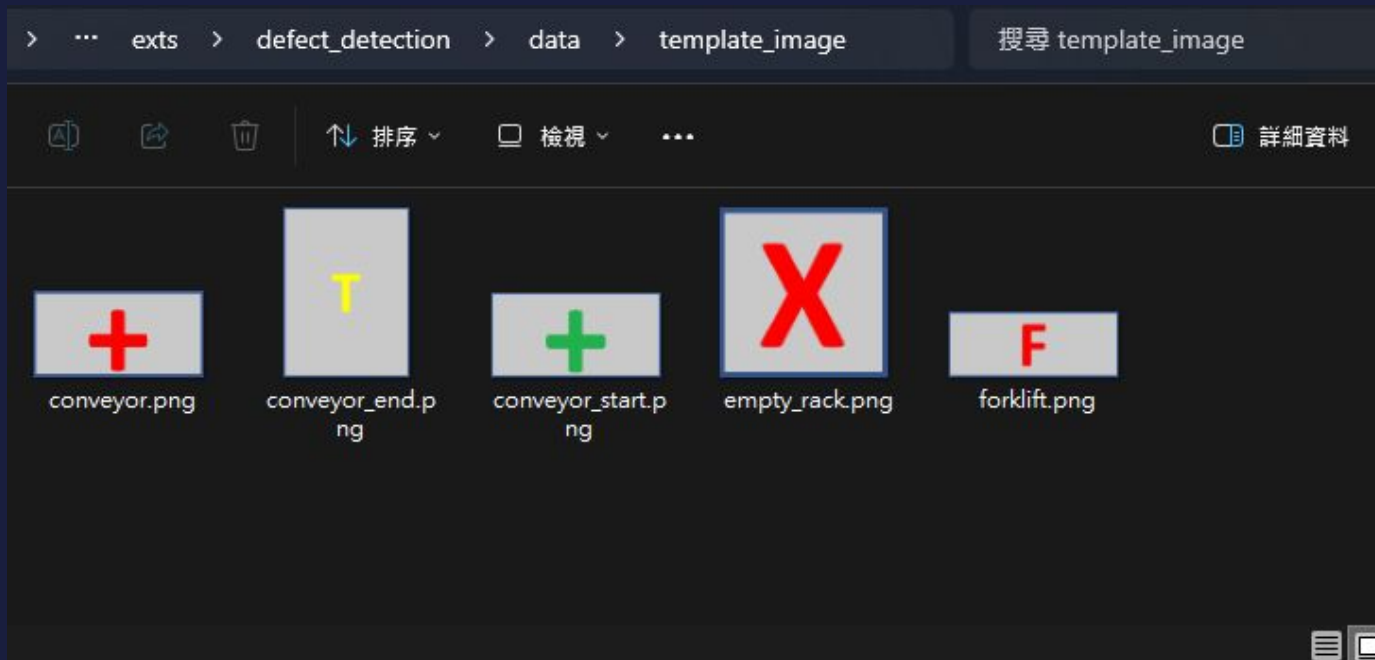
模擬瑕疵檢測系統(6/21)

因為要增加T字型輸送帶，對圖片做一下調整，
在輸送帶的最後，加上代表 T字型輸送帶的區塊。



模擬瑕疵檢測系統(7/21)

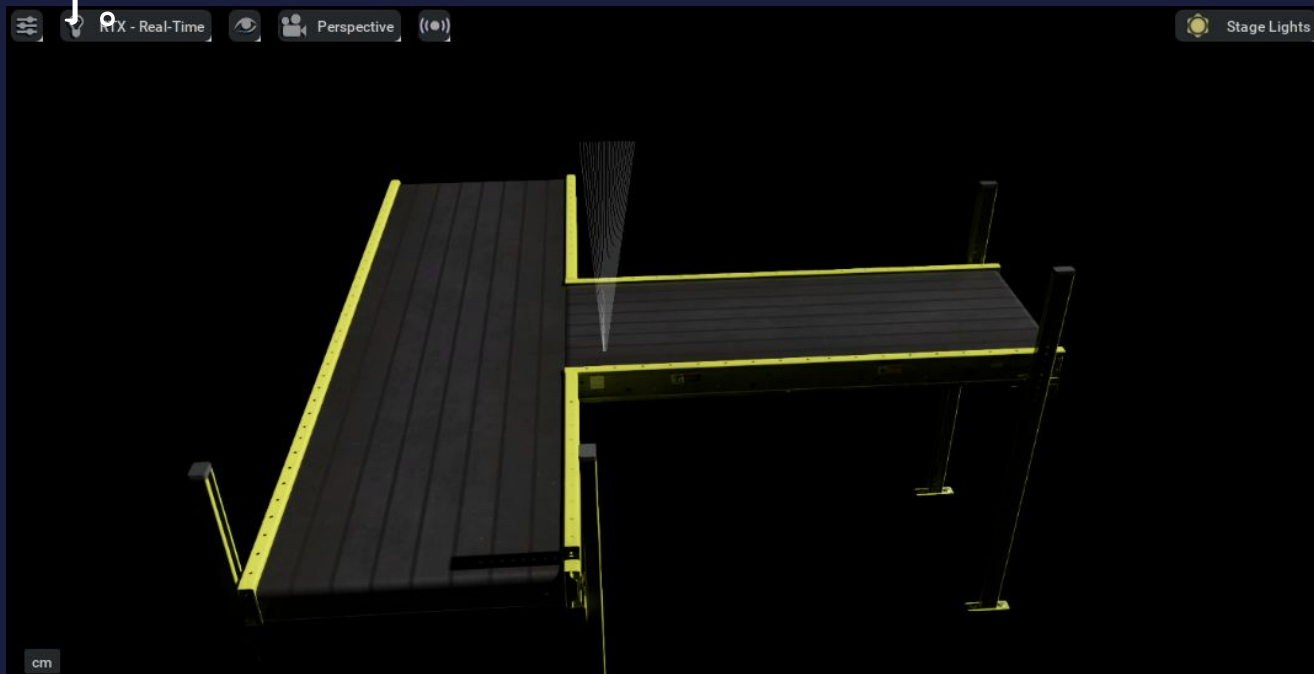
在template_image 加上T字型輸送帶區塊的圖片。



模擬瑕疵檢測系統(8/21)

接著準備 T 字型輸送帶的 USD 檔案，
首先加入一個 Lidar，用來偵測貨物是否到指定位置

了



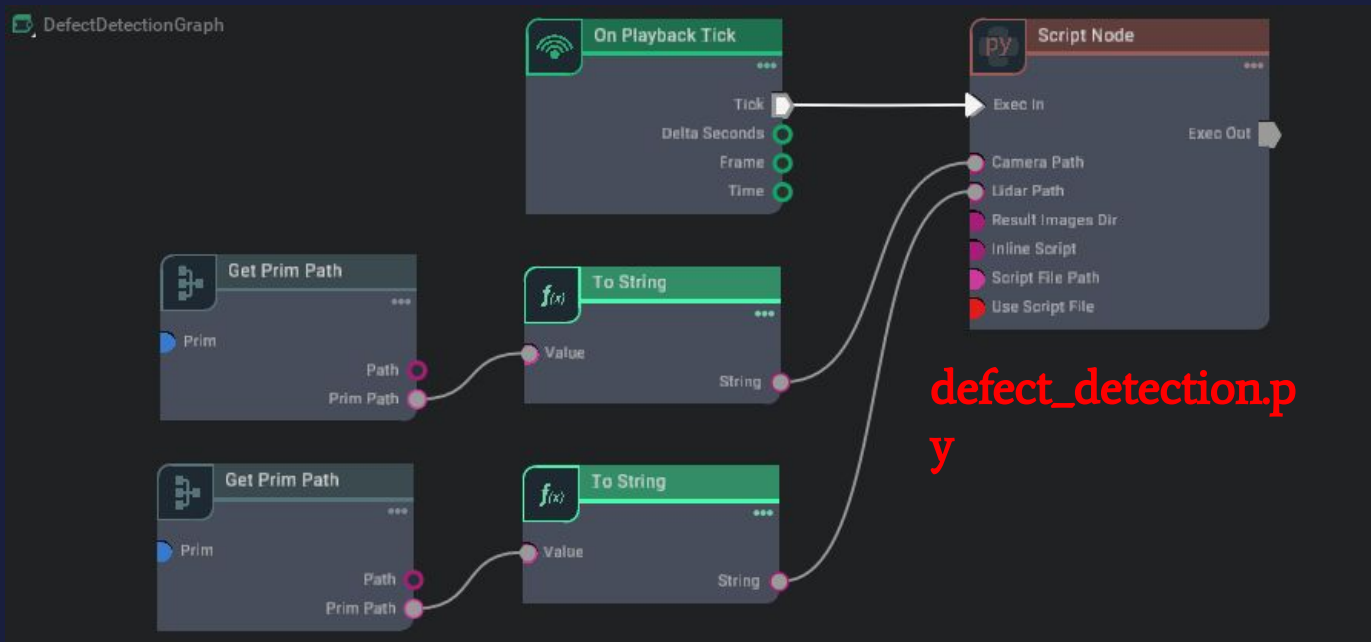
模擬瑕疵檢測系統(9/21)

接著加入 Camera , Camera 的畫面如下,
面向貨物輸送過來的方向。

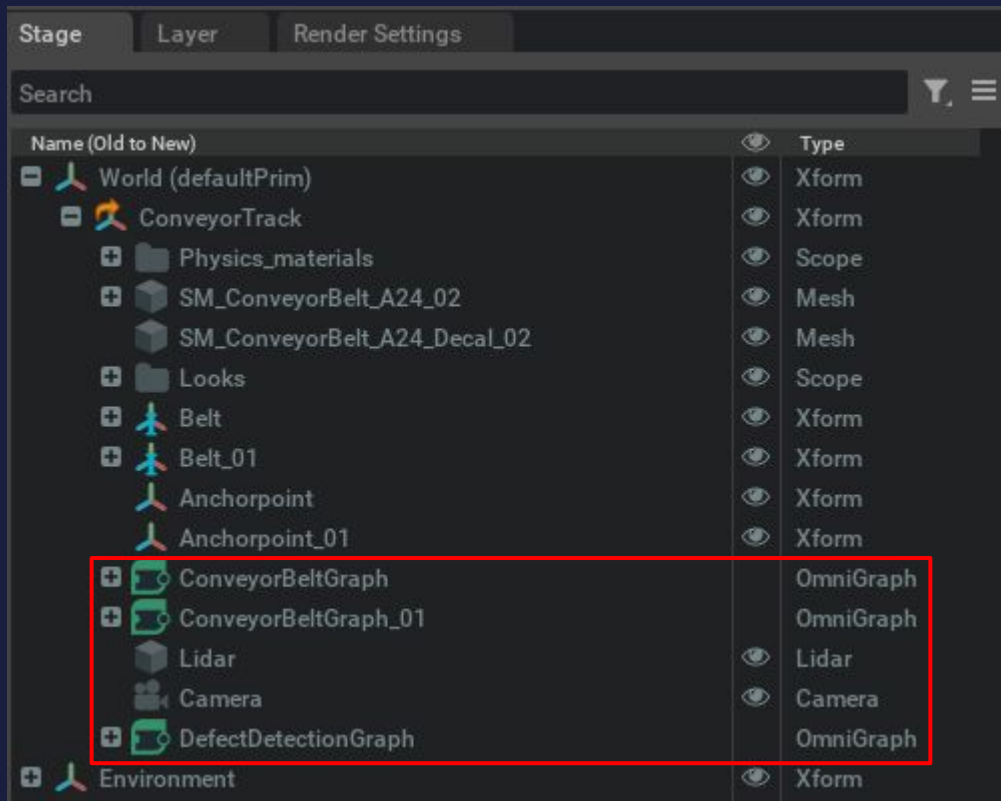


模擬瑕疵檢測系統(10/21)

在T字型輸送帶的 USD 檔內加上一個 Action Graph ,
這個 Action Graph 會把 Lidar 和 Camera 的路徑傳給 Script Node ,
Script Node 會不斷獲取 Lidar 資料, 確認貨物到位置就使用 Camera 拍照後做瑕疵檢測。



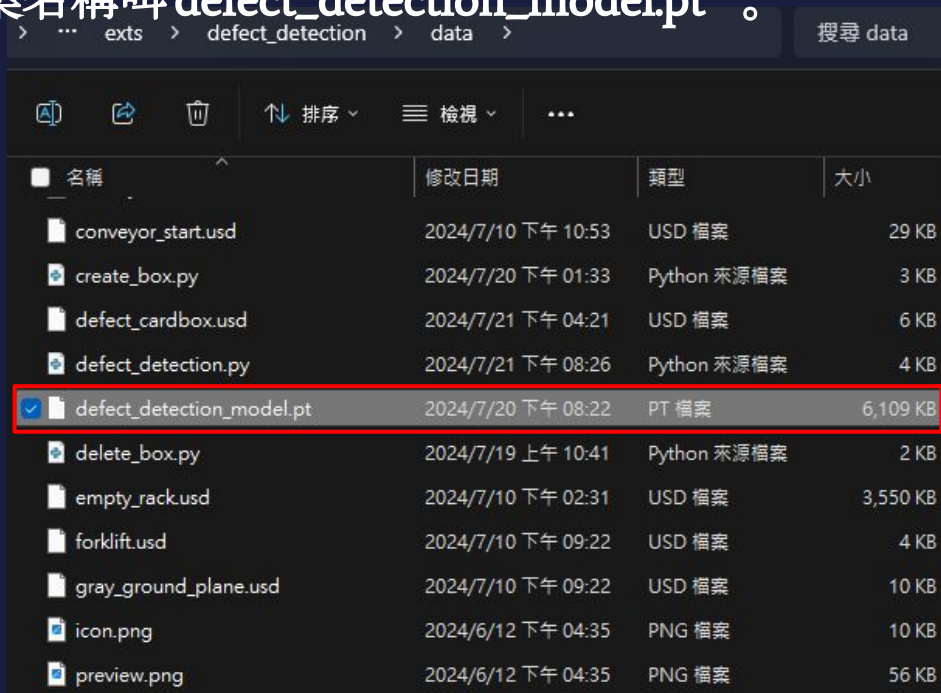
模擬瑕疵檢測系統(11/21)



所以T字型輸送帶的 USD 檔，
內容如左圖所示，
比較重要的是紅框標起來的部分，
後續程式碼會去讀取這幾個物件。

模擬瑕疵檢測系統(12/21)

瑕疵檢測用的模型已經事先訓練好放在 data 資料夾，檔案名稱叫 defect_detection_model.pt。



The screenshot shows a file explorer window with the path 'exts > defect_detection > data'. The search bar contains 'data'. The file list is as follows:

名稱	修改日期	類型	大小
conveyor_start.usd	2024/7/10 下午 10:53	USD 檔案	29 KB
create_box.py	2024/7/20 下午 01:33	Python 來源檔案	3 KB
defect_cardbox.usd	2024/7/21 下午 04:21	USD 檔案	6 KB
defect_detection.py	2024/7/21 下午 08:26	Python 來源檔案	4 KB
defect_detection_model.pt	2024/7/20 下午 08:22	PT 檔案	6,109 KB
delete_box.py	2024/7/19 上午 10:41	Python 來源檔案	2 KB
empty_rack.usd	2024/7/10 下午 02:31	USD 檔案	3,550 KB
forklift.usd	2024/7/10 下午 09:22	USD 檔案	4 KB
gray_ground_plane.usd	2024/7/10 下午 09:22	USD 檔案	10 KB
icon.png	2024/6/12 下午 04:35	PNG 檔案	10 KB
preview.png	2024/6/12 下午 04:35	PNG 檔案	56 KB

模擬瑕疵檢測系統(13/21)

defect_detection.py

瑕疵檢測的程式碼這邊，首先需要注意，後續用YOLO來做瑕疵檢測，這邊選用的工具是 ultralytics，記得先安裝一下。

(Isaac Sim 有內建Pytorch，所以不用額外安裝 Pytorch)

```
1  import omni.replicator.core as rep
2  from omni.isaac.range_sensor import _range_sensor
3  import omni.kit.commands
4  import os
5  from pxr import Sdf, Usd
6  import time
7  import numpy as np
8  import cv2
9  from datetime import datetime
10 from ultralytics import YOLO
```

模擬瑕疵檢測系統(14/21)

defect_detection.py

在setup的部分, 先準備好後續取得 Lidar 跟 Camera 資訊會需要的變數。

```
13 def setup(db: og.Database):
14     state = db.per_instance_state
15     camera_path = db.inputs.camera_path
16     render_product = rep.create.render_product(camera_path, resolution=(256, 256))
17
18     state.rgb_annotator = rep.AnnotatorRegistry.get_annotator("rgb")
19     state.rgb_annotator.attach([render_product])
20
21     state.lidarInterface = _range_sensor.acquire_lidar_sensor_interface()
22     state.lidarPath = db.inputs.lidar_path
23
24     state.is_triggered = False
25     state.last_time = time.time()
```


模擬瑕疵檢測系統(15/21)

defect_detection.py

接著準備後續調整 T字型輸送帶的速度需要用的變數，還有比較需要特別注意的，ultralytics 在進行第一次檢測會花比較長的時間，所以先在這邊進行一次 predict，後面在模擬開始後檢測就不會卡。

```
27 parent_path = "/".join(camera_path.split("/")[:-1])
28 state.conveyor_speed_property = f"{parent_path}/ConveyorBeltGraph/ConveyorNode.inputs:velocity"
29 state.conveyor_name = parent_path.split("/")[2]
30
31 manager = omni.kit.app.get_app().get_extension_manager()
32 extension_data_path = os.path.join(manager.get_extension_path_by_module("defect_detection"), "data")
33
34 # 第一次predict會花比較長時間，所以在這個階段先做完
35 model_path = os.path.join(extension_data_path, "defect_detection_model.pt")
36 state.model = YOLO(model_path)
37 warming_up_image = cv2.imread(os.path.join(extension_data_path, "warming_up.jpg"))
38 results = state.model(warming_up_image)
```

模擬瑕疵檢測系統(16/21)

defect_detection.p

compute 這邊，每一秒會去取得一次 Lidar 的資料，
當偵測到的距離小於一個門檻 值後，就會觸發 Camera 拍照的動作。

(is_triggered 是用來避免貨物在 Lidar 前面太久，導致重複觸發拍照)

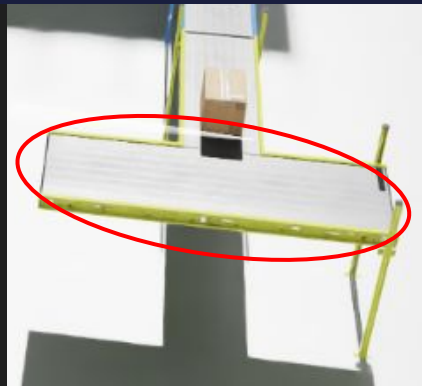
```
45 def compute_defect_detection():
46     state = db.per_instance_state
47     if time.time() - state.last_time > 1:
48         # 取得光達資料
49         depth = np.sum(state.lidarInterface.get_linear_depth_data(state.lidarPath))
50
51         if depth < 10 and state.is_triggered == False:
52             current_datetime = datetime.now().strftime("%Y%m%d_%H%M%S")
53
54             # 取得圖片資料
55             rgb = state.rgb_annotator.get_data(device="cuda").numpy()
56
57             # 轉為 BGR
58             bgr = cv2.cvtColor(rgb, cv2.COLOR_RGBA2BGR)
```

模擬瑕疵檢測系統(17/21)

defect_detection.p

接著就對 Camera 取得的圖片做檢測，如果圖片中沒有檢測到任何東西，
就會將T字型輸送帶紅框標示的那段，設定速度為正的 200，
並將圖片保存到 normal_images 資料夾。

```
60 results = state.model(bgr) # predict on an image
61 result = results[0]
62
63
64 # 處理預測結果
65 boxes = result.bboxes
66 objects_len = len(result)
67 if objects_len == 0:
68     omni.kit.commands.execute('ChangeProperty',
69                               prop_path=Sdf.Path(state.conveyor_speed_property),
70                               value=200,
71                               prev='')
72     image_save_dir = os.path.join(db.inputs.result_images_dir, "normal_images")
73     image_path = os.path.join(image_save_dir, f"{state.conveyor_name}-{current_datetime}.png")
```



模擬瑕疵檢測系統(18/21)

defect_detection.p

當檢測出瑕疵時，輸送帶則將速度設為負的 200，
並將瑕疵位置標註在圖片上，將圖片保存到 defect_images 資料夾。
(is_triggered 在拍照後會設為 True，直到貨物離開 Lidar 的範圍才會設回

```
False)
else:
    omni.kit.commands.execute('ChangeProperty',
        prop_path=Sdf.Path(state.conveyor_speed_property),
        value=-200,
        prev='')
    for i in range(objects_len):
        bounding_box = np.array(boxes.xyxy[i].cpu(), dtype=int)
        top_left = (bounding_box[0], bounding_box[1])
        bottom_right = (bounding_box[2], bounding_box[3])
        bgr = cv2.rectangle(bgr, top_left, bottom_right, (0, 0, 255), 3, cv2.LINE_AA)
        image_save_dir = os.path.join(db.inputs.result_images_dir, "defect_images")
        image_path = os.path.join(image_save_dir, f"{state.conveyor_name}-{current_datetime}.png")
        cv2.imwrite(image_path, bgr)

    state.is_triggered = True
elif depth >= 10:
    state.is_triggered = False
```

模擬瑕疵檢測系統(19/21)

extension.p

y

在Extension 主程式的部分也需要做一些修改，
當使用者按下 Load Scene 之後，會建立用來存放瑕疵檢測圖片的資料夾。

```
29 def on_click():
30     # Read image
31     image = cv2.imread(string_model.as_string)
32     stage_w, stage_h = image.shape[1], image.shape[0]
33
34     # Create result images directory
35     dir_name = os.path.dirname(string_model.as_string)
36     result_images_dir = os.path.join(dir_name, "result_images")
37     if os.path.exists(result_images_dir):
38         shutil.rmtree(result_images_dir)
39     os.mkdir(result_images_dir)
40     os.mkdir(os.path.join(result_images_dir, "normal_images"))
41     os.mkdir(os.path.join(result_images_dir, "defect_images"))
```


模擬瑕疵檢測系統(20/21)

extension.p

y

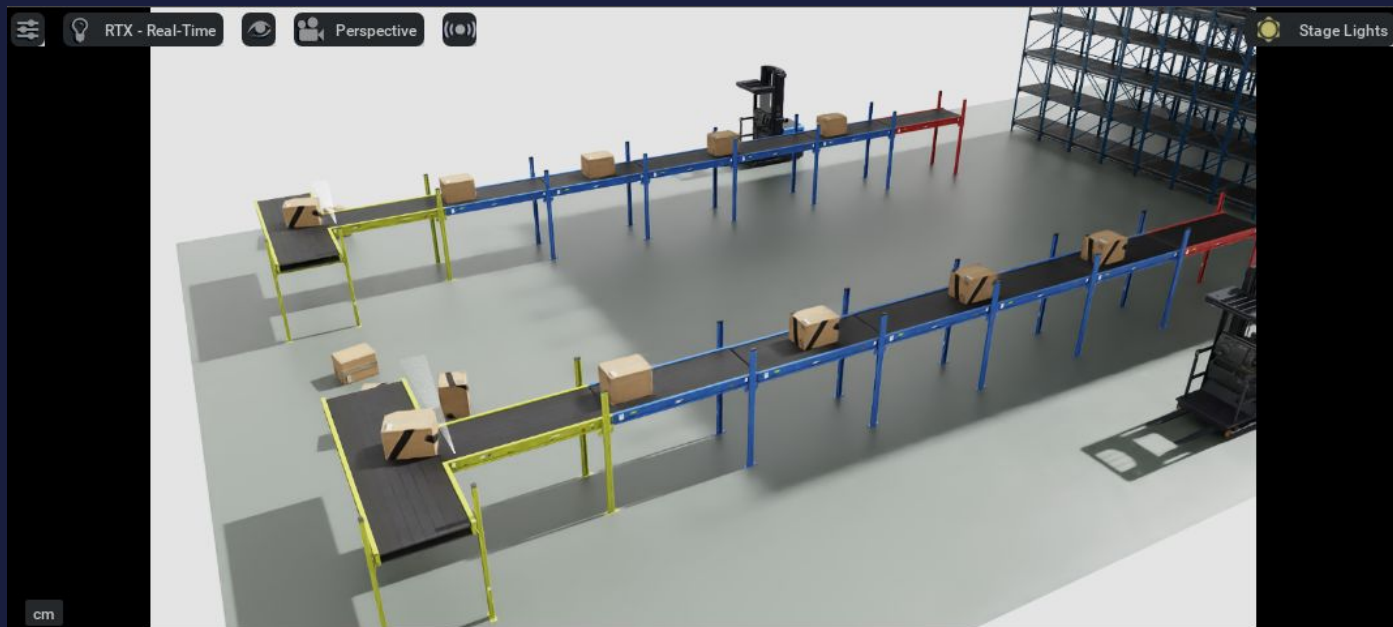
然後在生成 T 字型輸送帶時，需要設定瑕疵檢測程式碼的路徑，
以及把存放瑕疵檢測圖片用的資料夾路徑，
設定給 script_node 的 result_images_dir 。

```
175 elif image_name[:-4] == 'conveyor_end':
176     script_path_property = f"{prim_path}/ConveyorTrack/DefectDetectionGraph/script_node.inputs:scriptPath"
177     omni.kit.commands.execute('ChangeProperty',
178                               prop_path=Sdf.Path(script_path_property),
179                               value=os.path.join(extension_data_path, 'defect_detection.py'),
180                               prev='')
181
182     result_images_dir_property = f"{prim_path}/ConveyorTrack/DefectDetectionGraph/script_node.inputs:result_images_dir"
183     omni.kit.commands.execute('ChangeProperty',
184                               prop_path=Sdf.Path(result_images_dir_property),
185                               value=result_images_dir,
186                               prev='')

```

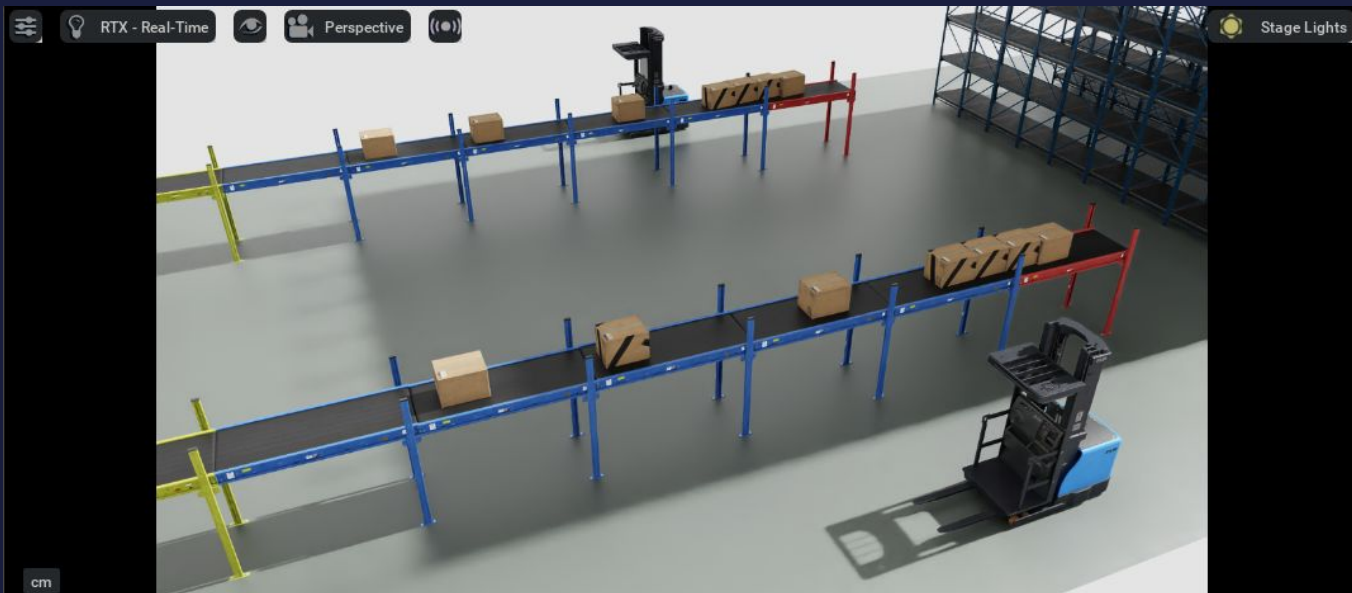
模擬瑕疵檢測系統(21/21)

到這邊就完成了，可以測試一下運行的狀況。
(特別注意一點，在執行過程中最好不要去點其他視窗，
可能會導致 Omniverse 的運行變慢。)



透過外部通訊的方式，控制場景的運作(1/9)

繼續給場景加上一些額外功能，
我們想讓場景能透過某種通訊方式，讓外部有辦法控制場景的運作，
像是從外部發一個訊號，讓所有藍色輸送帶停下來。

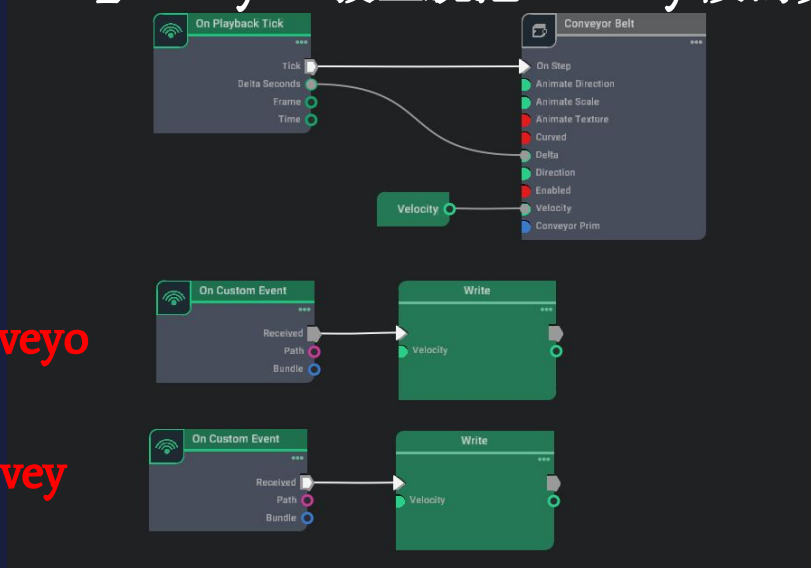


透過外部通訊的方式，控制場景的運作(2/9)

修改藍色輸送帶的 USD 檔，在 ConveyorBeltGraph 的地方

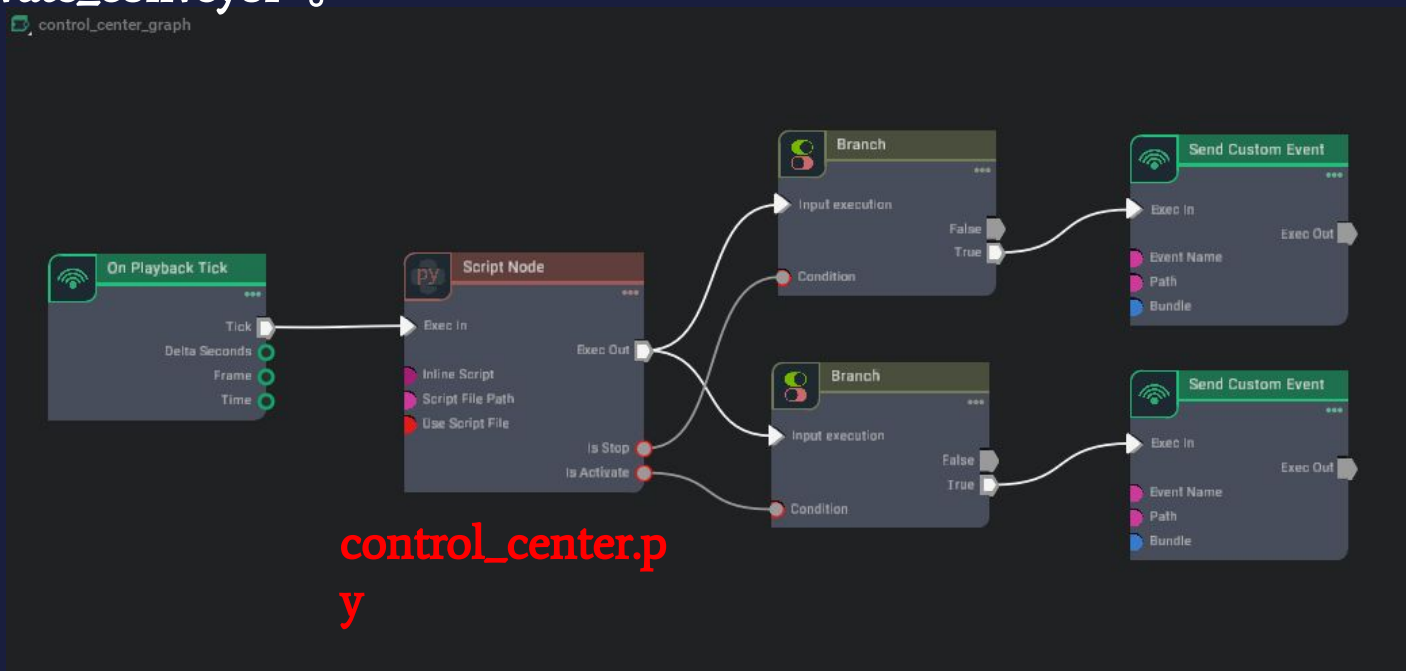
，
新增一個變數 Velocity 用來控制輸送帶的速度，
當事件 stop_conveyor 發生就把 Velocity 設為 0，
當事件 activate_conveyor 發生就把 Velocity 設為負的 100。

stop_conveyor
or
activate_conveyor



透過外部通訊的方式，控制場景的運作(3/9)

接著我們想在場景中有一個 Server 不斷接受使用者從外部傳進來的訊息，根據接受到的訊息，決定要觸發 `stop_conveyor` 事件還是 `activate_conveyor`。



透過外部通訊的方式，控制場景的運作(4/9) control_center.py

為了接收外部傳進來的資訊，這裡使用 Socket 的方式來溝通

，
先在setup的上方定義ID、Port和此後會用到的變數。

```
42 def setup(db: og.Database):  
43     state = db.per_instance_state  
44  
45     state.host = '0.0.0.0'  
46     state.port = 8080  
47  
48     state.socket_flag = False  
49     state.server_socket_thread = None  
50  
51     state.stop_conveyor_flag = False  
52     state.activate_conveyor_flag = False  
53  
54  
55 def cleanup(db: og.Database):  
56     state = db.per_instance_state  
57     state.socket_flag = False
```

透過外部通訊的方式，控制場景的運作(5/9)

control_center.py

compute 這邊會去開啟一個執行緒，
新開啟的執行緒負責不斷的確證有沒有使用者連線進來

，
並根據使用者傳輸的資訊，
對stop_conveyor_flag 和activate_conveyor_flag 做修改。

```
59 def compute(db: og.Database):
60     state = db.per_instance_state
61
62     if state.socket_flag == False:
63         state.socket_flag = True
64         state.server_socket_thread = threading.Thread(target=start_server_socket, args = (state.host, state.port, db
65         state.server_socket_thread.start()
66
67     if state.stop_conveyor_flag:
68         db.outputs.is_stop = True
69         state.stop_conveyor_flag = False
70     else:
71         db.outputs.is_stop = False
72
73     if state.activate_conveyor_flag:
74         db.outputs.is_activate = True
75         state.activate_conveyor_flag = False
76     else:
77         db.outputs.is_activate = False
```

透過外部通訊的方式，控制場景的運作(6/9)

control_center.py

```
20 def start_server_socket(host, port, db):
21     state = db.per_instance_state
22
23     server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
24     server_socket.bind((host, port))
25     server_socket.listen(1)
26     server_socket.settimeout(1)
27
28     while state.socket_flag:
29         try:
30             conn, addr = server_socket.accept()
31         except socket.timeout:
32             continue
33         recv_data = conn.recv(1024)
34         recv_data = recv_data.decode()
35         print('recv_data: ' + recv_data)
36
37         if recv_data == "stop_conveyor":
38             state.stop_conveyor_flag = True
39         elif recv_data == "activate_conveyor":
40             state.activate_conveyor_flag = True
41
42         conn.send(recv_data.encode())
43         conn.close()
44
45     server_socket.close()
```

當使用者傳輸 stop_conveyor 字串，
就把stop_conveyor_flag 設為True。

當使用者傳輸 activate_conveyor 字串，
就把activate_conveyor_flag 設為True。

透過外部通訊的方式，控制場景的運作(7/9)

extension.p

最後，在 extension.py 多一段程式碼用來生成上述的 Action Graph。

```
88 # Create control center action graph
89 graph_path = f"/control_center_graph"
90 keys = og.Controller.Keys
91 (graph_handle, list_of_nodes, _, _) = og.Controller.edit(
92     {"graph_path": graph_path, "evaluator_name": "execution"},
93     {
94         keys.CREATE_NODES: [
95             ("on_playback_tick", "omni.graph.action.OnPlaybackTick"),
96             ("script_node", "omni.graph.scriptnode.ScriptNode"),
97             ("stop_branch_node", "omni.graph.action.Branch"),
98             ("activate_branch_node", "omni.graph.action.Branch"),
99             ("send_stop_event_node", "omni.graph.action.SendCustomEvent"),
100             ("send_activate_event_node", "omni.graph.action.SendCustomEvent")
101         ],
102         keys.CREATE_ATTRIBUTES: [
103             ("script_node.outputs:is_stop", "bool"),
104             ("script_node.outputs:is_activate", "bool"),
105         ],
106         keys.SET_VALUES: [
107             ("script_node.inputs:usePath", True),
```


透過外部通訊的方式，控制場景的運作(8/9)

另外準備了一份程式碼，用來當作外部的使用者，後續就用這份程式碼傳輸資訊到 Omniverse 內。

**conveyor_controller.
py**

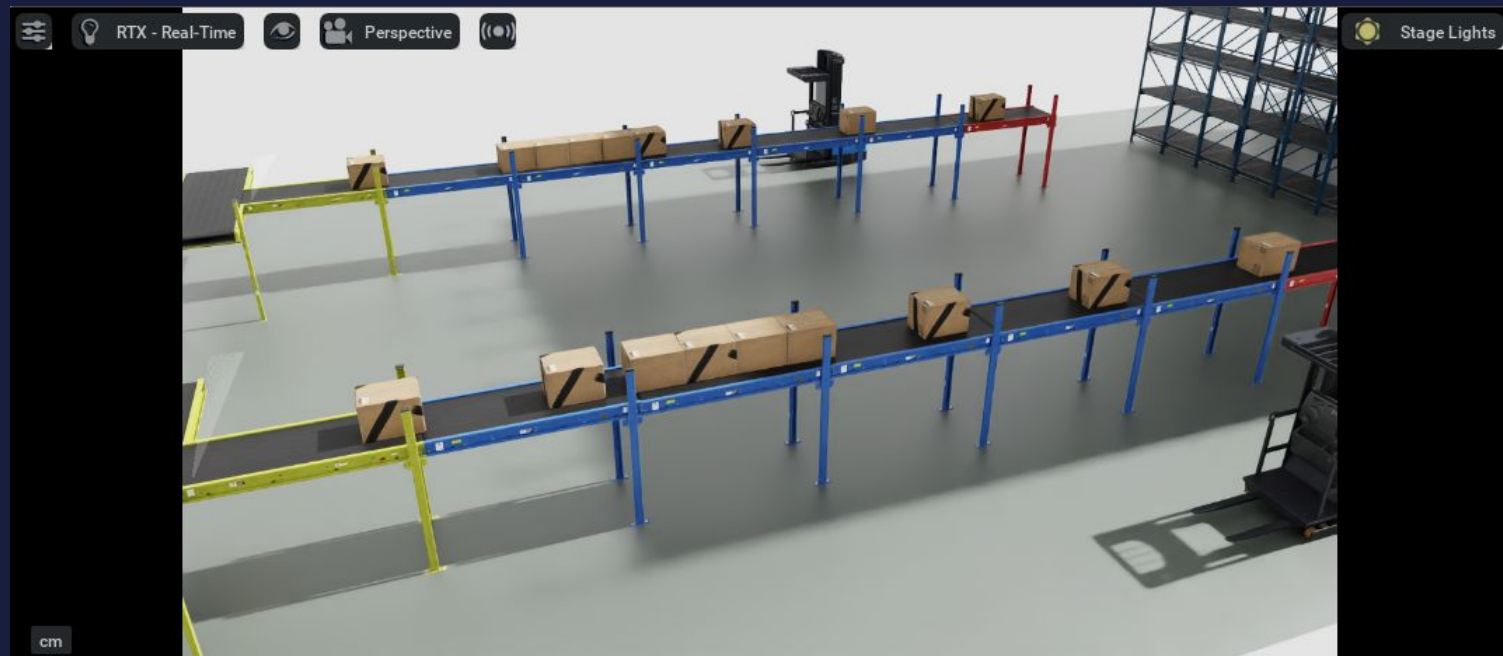
```
conveyor_controller.py X
C: > Users > rt > Desktop > Jerry_Zone > omniverse_related > Lesson_Related > Day7 > conveyor_controller.py > ...
1 import socket
2
3 HOST = '127.0.0.1'
4 PORT = 8080
5
6 client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7 client_socket.connect((HOST, PORT))
8
9 send_data = 'stop_conveyor'
10 # send_data = 'activate_conveyor'
11
12 print('send: ' + send_data)
13 client_socket.send(send_data.encode())
14
15 recv_indata = client_socket.recv(1024)
16 print('recv: ' + recv_indata.decode())
17
18 client_socket.close()
```

要停止輸送帶，
send_data 就設 stop_conveyor 。

要啟動輸送帶，
send_data 就設
activate_conveyor 。

透過外部通訊的方式，控制場景的運作(9/9)

用上面那份測試用程式碼，確認一下可不可以控制輸送帶停止和 啟動。



練習時間二

基礎題：

試著用執行緒的方式，每隔五秒生成一個 Cube。

進階題：

用任一通訊方式，從外部控制輸送帶停止。

Coroutine(1/2)

如果是要同時執行多件事，除了前面提到的 `Thread` 之外，也可以考慮使用 `Coroutine`，以練習題二的每 5 秒生成一個物件為例，可以寫成如下的程式碼。

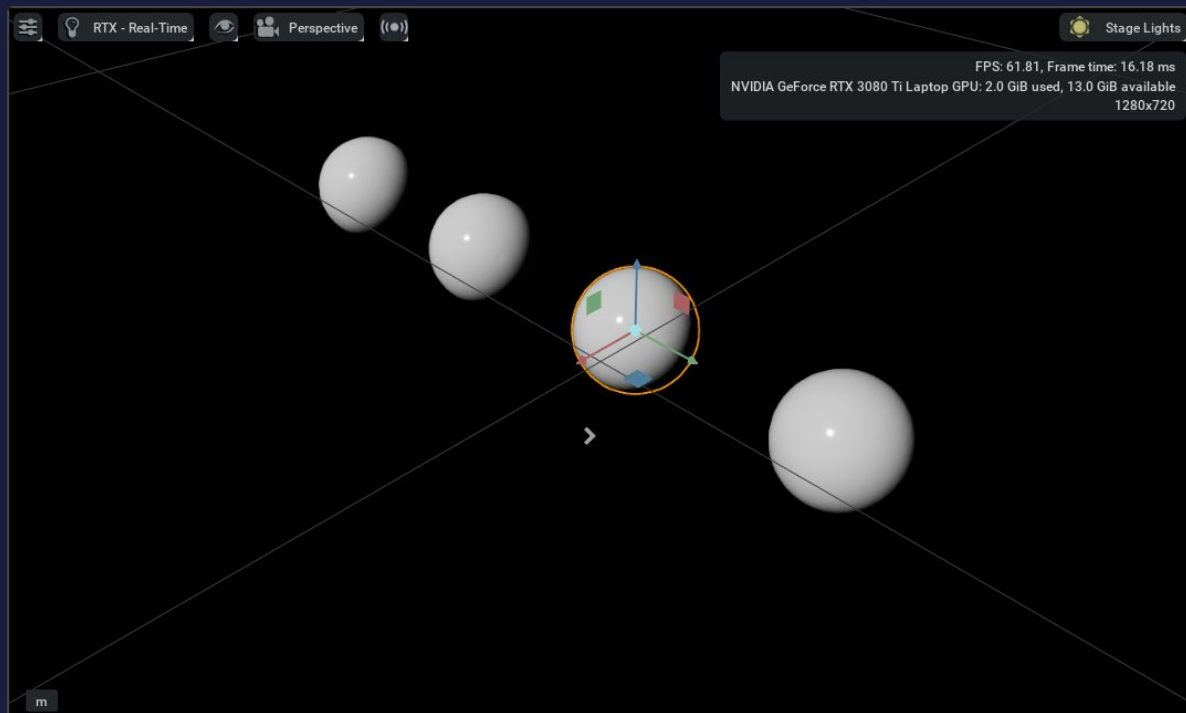
```
12 import omni.kit.commands
13 import asyncio
14
15 async def create_sphere(state):
16     while state.flag:
17         omni.kit.commands.execute('CreateMeshPrimWithDefaultXform',
18                                   prim_type='Sphere',
19                                   above_ground=True)
20         await asyncio.sleep(5)
21
22
23 def setup(db: og.Database):
24     state = db.per_instance_state
25     state.flag = True
26     asyncio.ensure_future(create_sphere(state))
27
28
29 def cleanup(db: og.Database):
30     state = db.per_instance_state
31     state.flag = False
```

特別注意無限迴圈
一定要給一個終止條件，
以這邊來說就是用 `state.flag` 來控制。

在 `cleanup` 的地方記得要讓
`state.flag` 設為 `False`，
不然就算切換到其他 USD 檔，
還是會不斷生成物件。

Coroutine(2/2)

按開始模擬後，每隔五秒就會有物件生成，並且不會卡住你的介面。



練習時間三

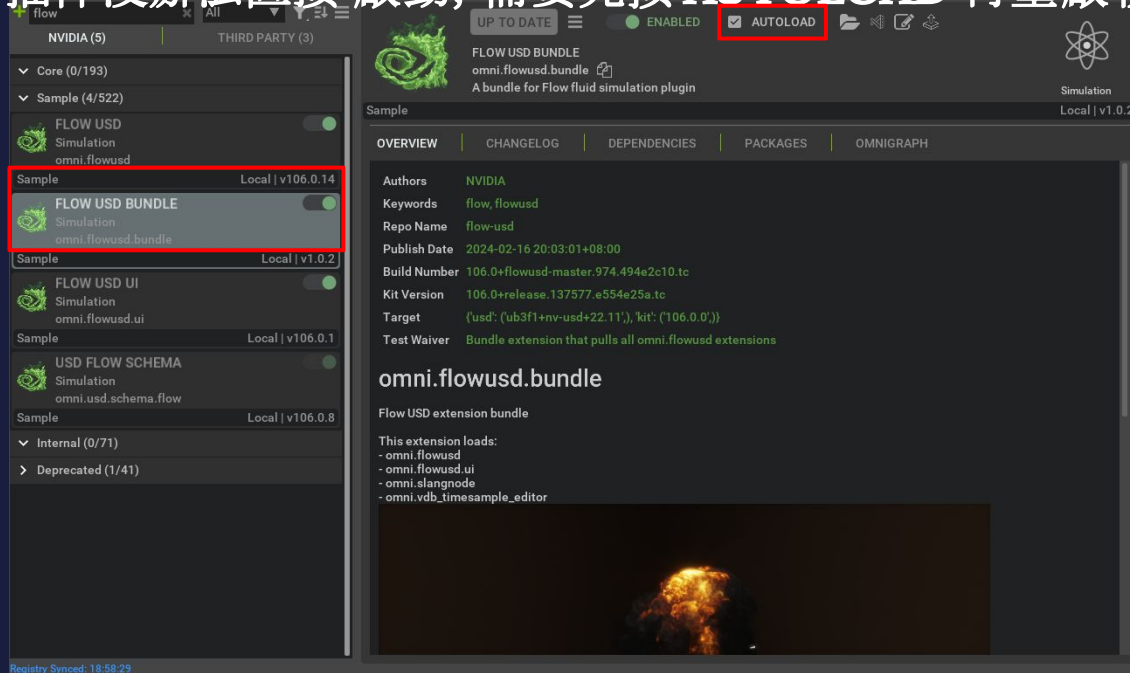
1. 將練習題二生成物件的程式碼改用 `Coroutine` 的方式寫。
2. 當使用者按下模擬停止時就不要再生成物件，
且當按下模擬開始時又會繼續生成物件。

補充

1. 火焰及煙霧
2. 程式碼參考網址

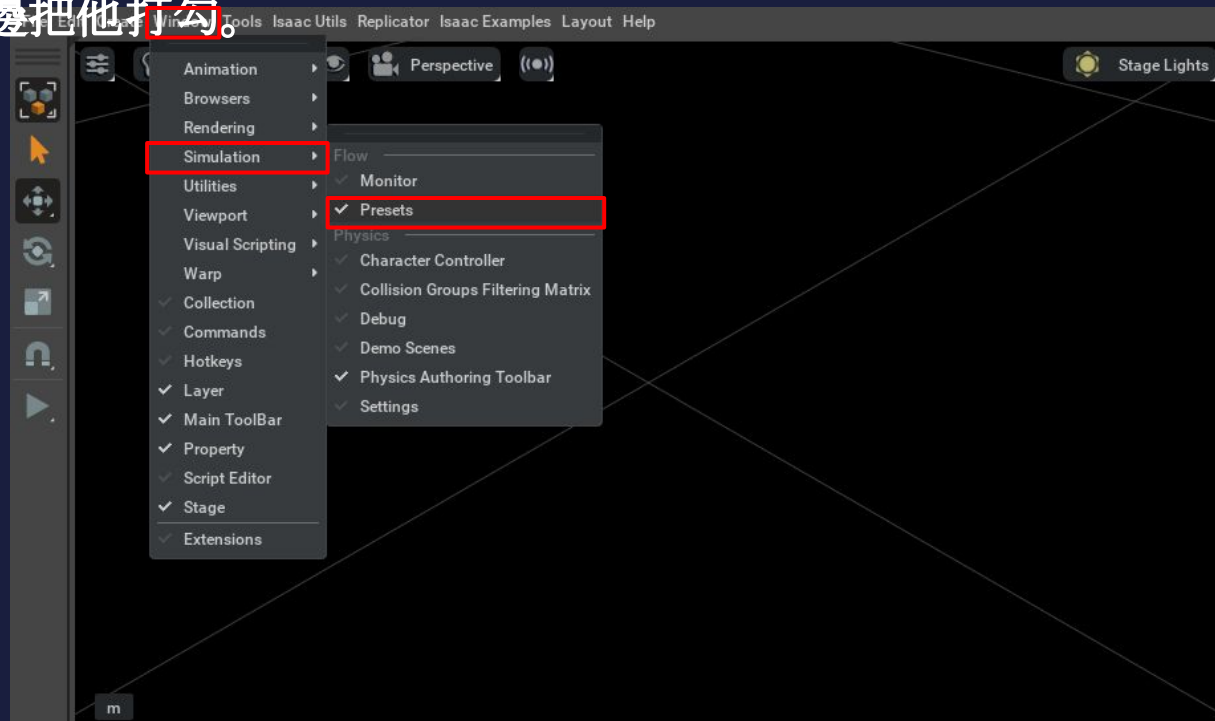
火焰及煙霧(1/13)

要生成火焰或煙霧，可以使用 FLOW USD BUNDLE 這個插件，
這個插件沒辦法直接啟動，需要先按 AUTOLOAD 再重啟軟體。



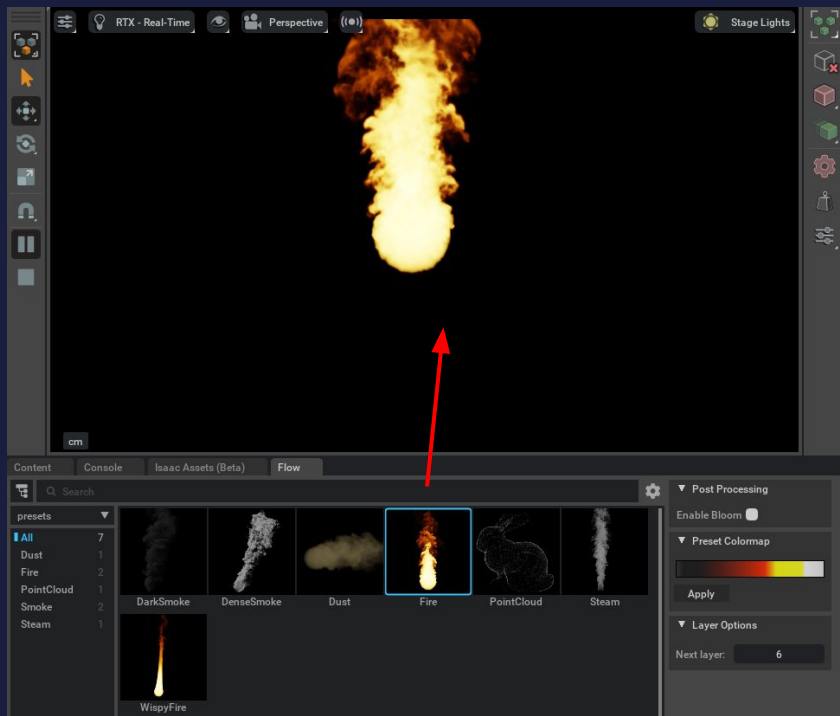
火焰及煙霧(2/13)

有開啟插件的話, 就可以在 Window -> Simulation 下找到 Presets , 這邊把他打勾。

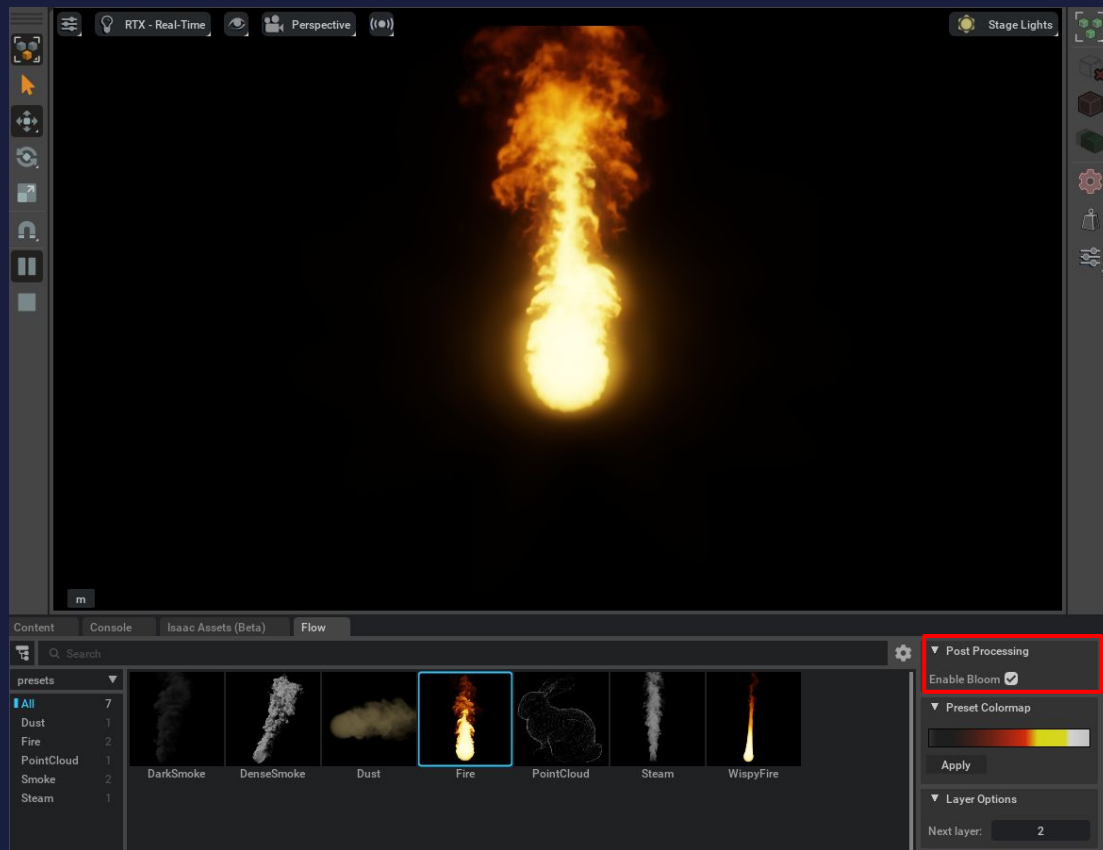


火焰及煙霧(3/13)

把火焰拖拉到場景中，再按下左側的開始模擬按鈕，就會看到火焰在燃燒了。



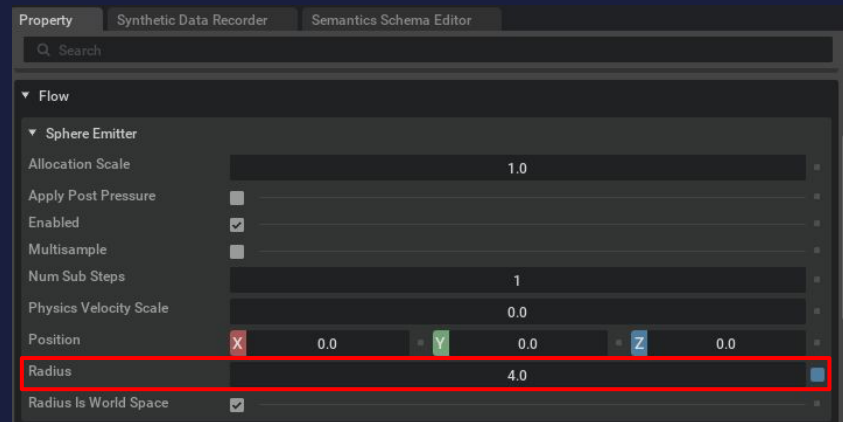
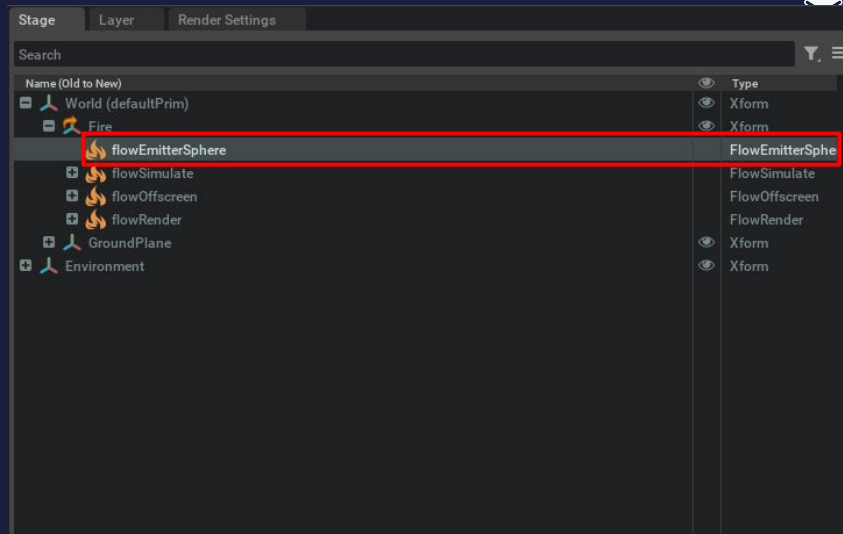
火焰及煙霧(4/13)



打勾右下角的 Enable Bloom ,
就會有光暈的效果,
看起來會比較真實一點。

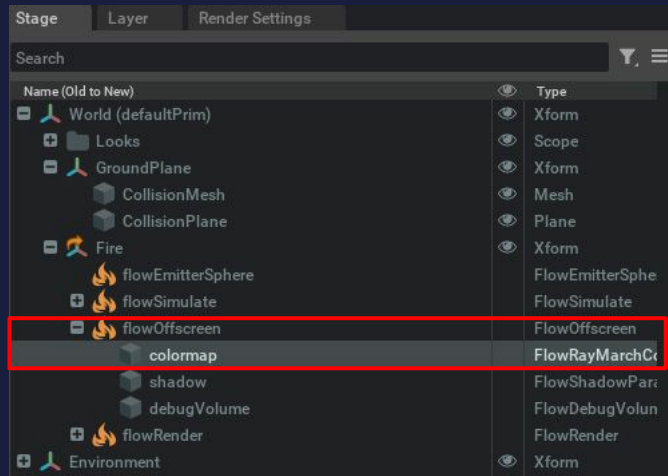
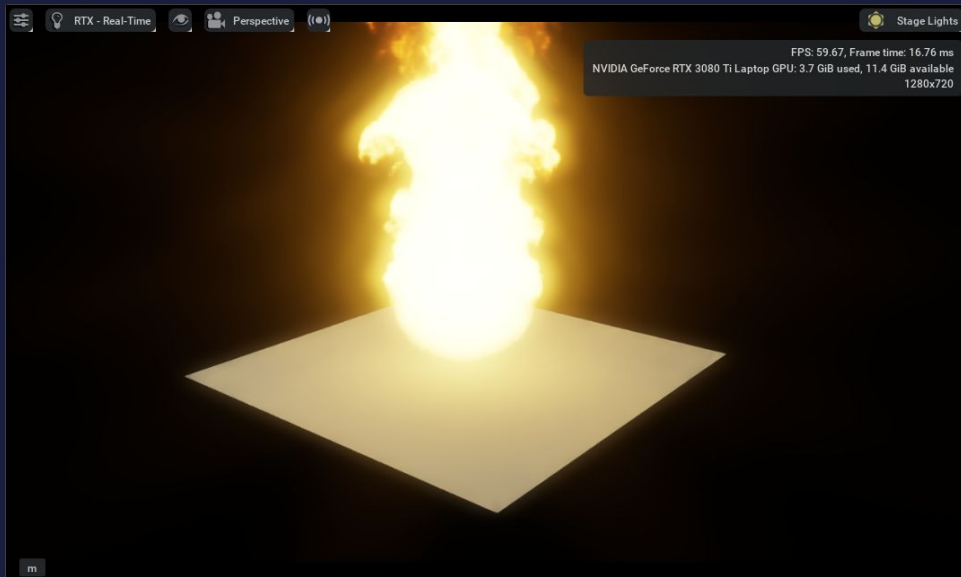
火焰及煙霧(5/13)

要調整火焰的大小，
不能直接調 Xform 的Scale，
需要調整 flowEmitterSphere 下的
Radius。



火焰及煙霧(6/13)

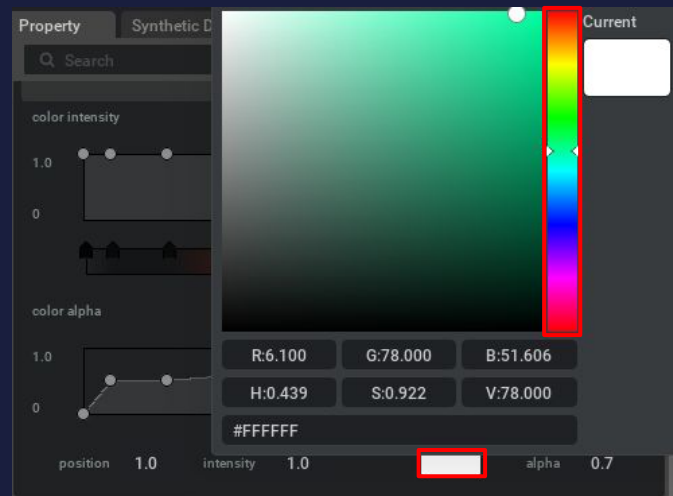
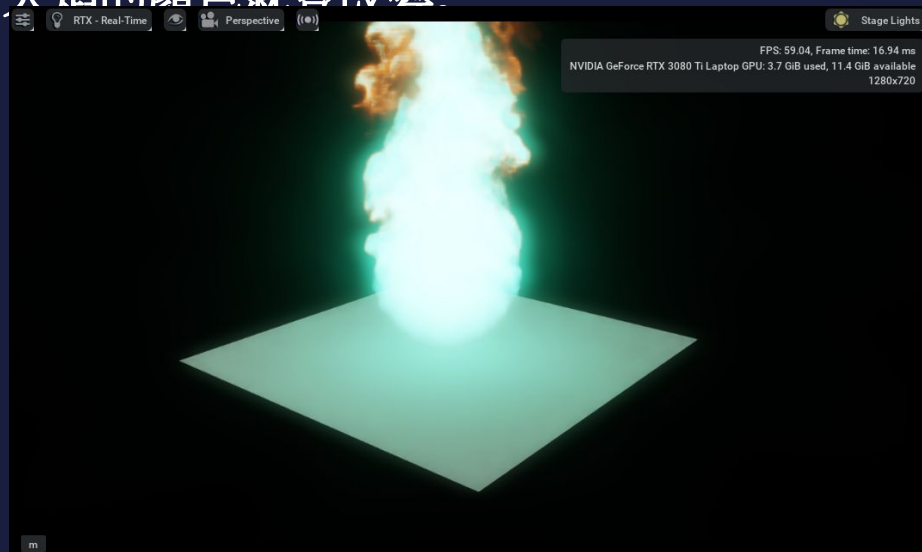
在flowOffscreen -> colormap 下，
將Color Scale 調高，可以讓火焰整體變
亮。



火焰及煙霧(7/13)

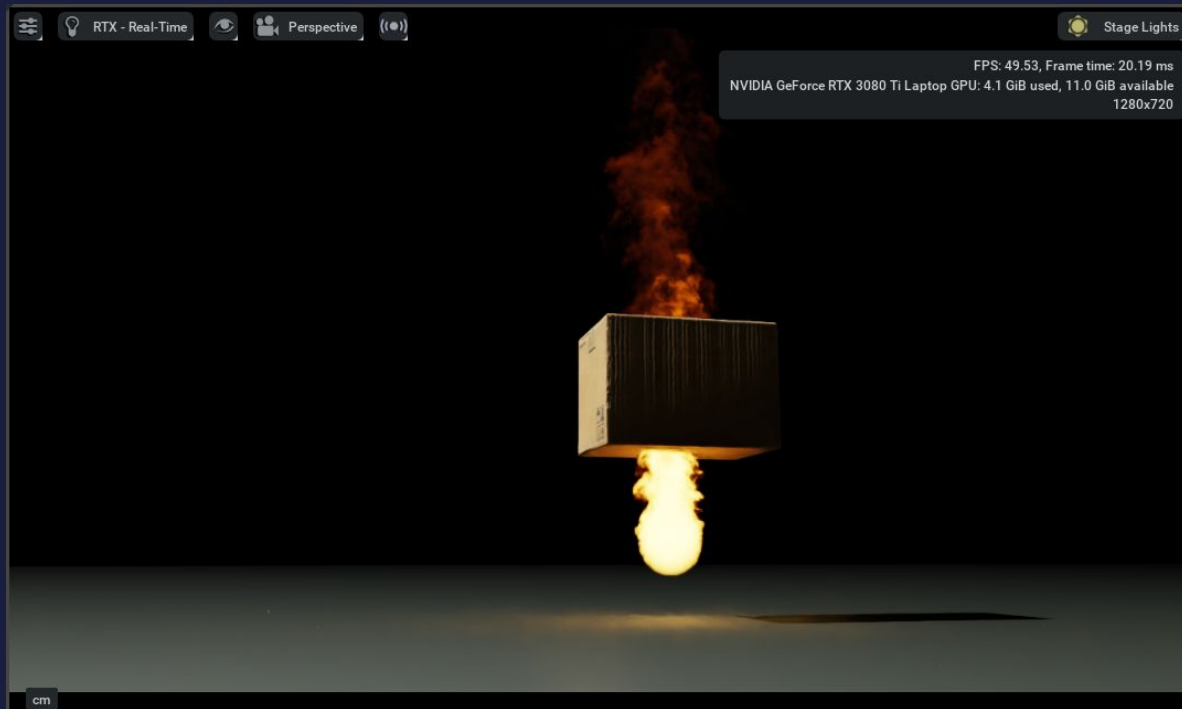
同樣在flowOffscreen -> colormap 下，
先點一下 color intensity 最右側的白色指標

，
再到顏色選擇器改動顏色，
火焰的顏色就會改變



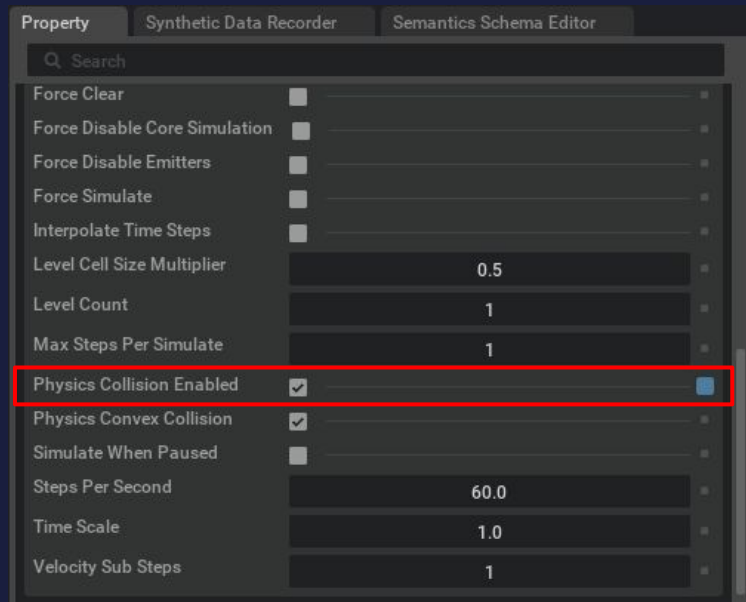
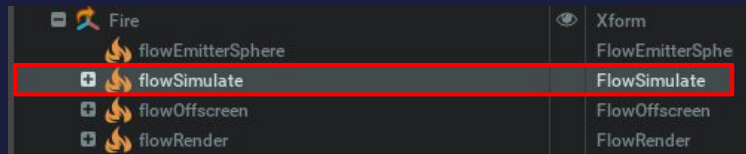
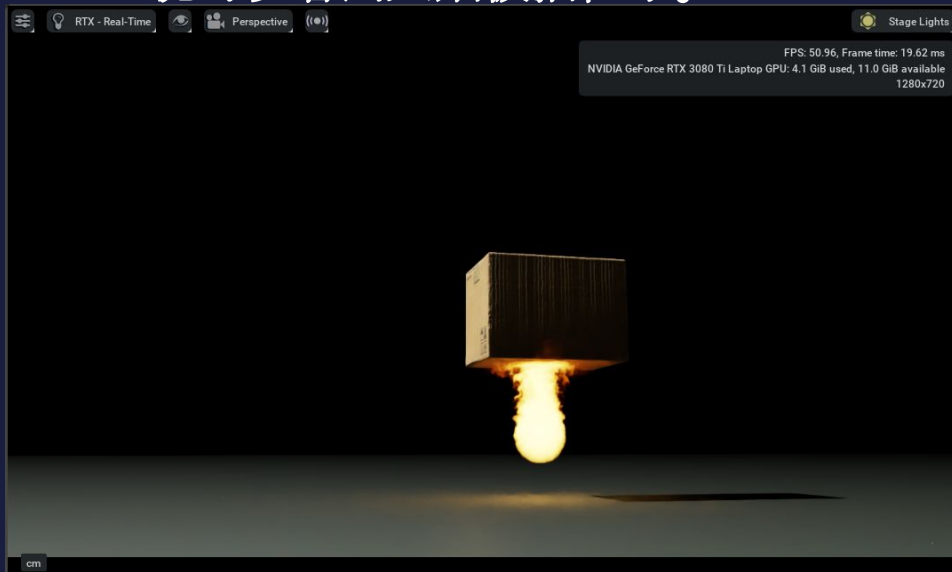
火焰及煙霧(8/13)

預設的情況下，火焰不會與其他物件 產生碰撞，
所以放障礙物在火焰上面，會看到火焰直接穿過去。



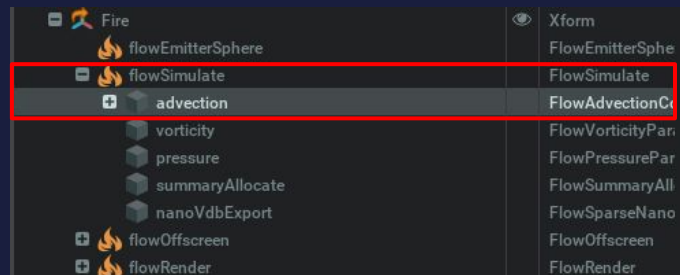
火焰及煙霧(9/13)

把flowSimulate 下的
Physics Collision Enabled 打勾
,
就可以看到火焰被擋住了。



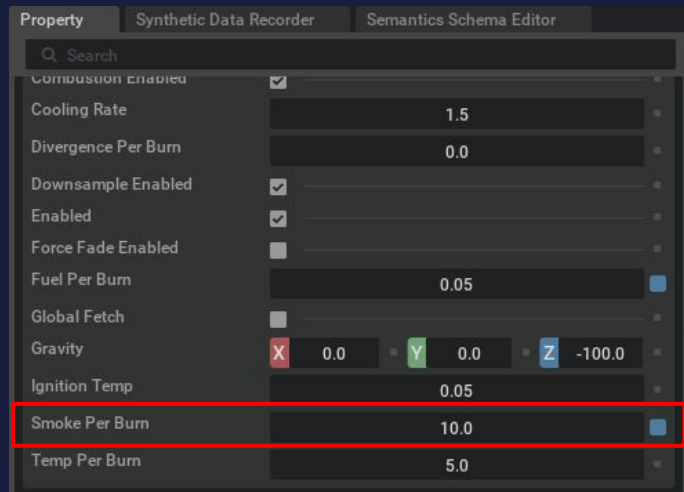
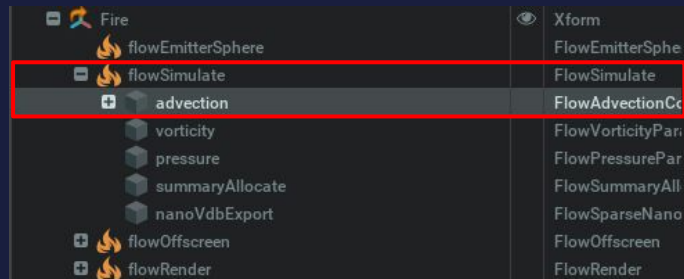
火焰及煙霧(10/13)

可以試著把火調大，可以更好看到碰撞的效果，
到flowSimulate -> advection 把
Fuel Per Burn 調小，火就會變大。



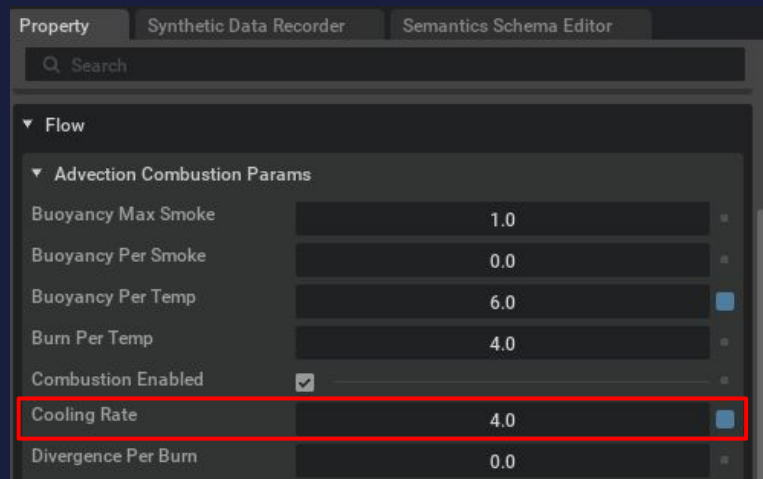
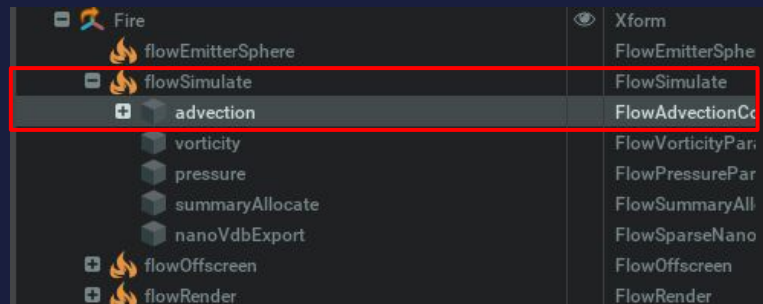
火焰及煙霧(11/13)

为了更好的看到煙，先讓背景是白色的，
然後到flowSimulate -> advection 把
Smoke Per Burn 調大，煙就會變大。



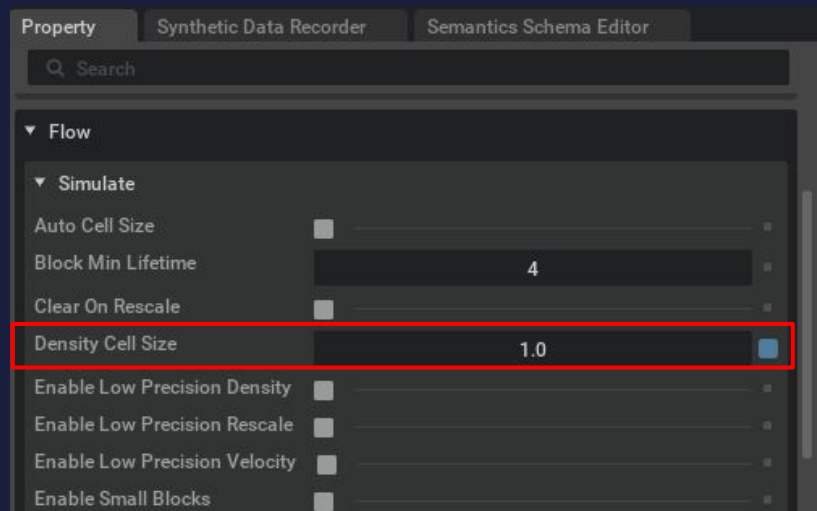
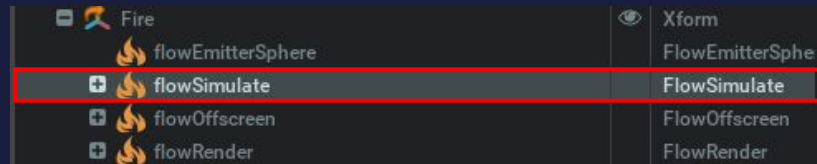
火焰及煙霧(12/13)

如果想要煙比較快消失，
可以到flowSimulate -> advection 把
Cooling Rate 調大，煙的存活時間就會比較短。



火焰及煙霧(13/13)

如果想要提升 FPS，
可以調整 flowSimulate 下的 Density Cell
Size，
調大的話火焰就會比較少細節，
但是效能會提升很多。



程式碼參考網址

<https://docs.ultralytics.com/quickstart/#use-ultralytics-with-python>

<https://forums.developer.nvidia.com/t/multi-threading-in-omniverse-extensions/236553>

<https://realpython.com/python-sockets/>