Soccer on Your Tabletop
Supplementary Material

Intercepting the GPU calls between the video game engine and the graphics card using RenderDoc [1] usually results in a massive amount of information, from texture and depth buffers to blend shape weights of the participating human models. While it is easy to locate and extract the color/depth buffers, the other information is more difficult to parse. For example, the naming of the intercepted variables is not always meaningful and in order to use them more sophisticated methods are needed [4].

In our case, the area of action (soccer field) has specific dimensions and structure that can be used to infer information such as camera matrices. In particular, our goal is to estimate the video game camera modelview matrix M_{mv} and projection matrix M_{proj} (OpenGL/DirectX), so we can invert the depth buffer that it is in Normalized Device Coordinates (NDC). Below (Appendix A) we describe the method to recover these matrices. The additional information that we need is a) a person segmentation of the texture buffer and b) an auxiliary camera that observes the field in world coordinates. To find which pixels belong to the players we use the semantic segmentation network of [7]. Then, for the auxiliary camera we follow the same approach as in Sec. 4.1: we estimate a camera with parameters M_{aux} that observes the soccer field that lies in the world center with y axis 0. Note that the intrinsics and extrinsics of the auxiliary camera cannot be used directly for estimating M_{proj} and M_{mv}, since they do not include the near/far plane parameters and we do not know the video game's world system.

Appendix A. Estimating Video Game Cameras

To estimate the camera parameters (modelview and projection matrices) of a video game image, we need its corresponding NDC buffer, the auxiliary camera M_{aux} that observes the field and the player segmentation mask.

The modelview and projection matrices in OpenGL/DirectX are 4×4 and the parameters to be estimated are the rotation vector $\theta_x, theta_y, theta_z$, the translation vector t_x, t_y, t_z and focal length, near and far plane $f, z_{\text{near}}, z_{\text{far}}$.

Essentially we want to find the parameters that transformed the world scene to NDC so we can invert them:

$$\begin{pmatrix} x_{\text{clip}} \\ y_{\text{clip}} \\ z_{\text{clip}} \\ w_{\text{clip}} \end{pmatrix} = M_{\text{proj}}\, M_{\text{mv}} \begin{pmatrix} x_{\text{world}} \\ y_{\text{world}} \\ z_{\text{world}} \\ 1 \end{pmatrix} \tag{1}$$

$$\begin{pmatrix} x_{\text{NDC}} \\ y_{\text{NDC}} \\ z_{\text{NDC}} \end{pmatrix} = \begin{pmatrix} x_{\text{clip}}/w_{\text{clip}} \\ y_{\text{clip}}/w_{\text{clip}} \\ z_{\text{clip}}/w_{\text{clip}} \end{pmatrix} \tag{2}$$

To estimate the OpenGL cameras we can rely on the structure of the scene: points that belong to the soccer field (and not to players) should lie on a plane with y equals to 0. The 3D world position of the ground pixels can be found by intersecting the rays from the auxiliary camera's ground pixels towards the ground. However, if we optimize only for the ground pixels, the $z_{\text{near}}, z_{\text{far}}$ parameters collapse, resulting in wrong reconstructions for the players. In addition to the ground constraint, we can minimize the reprojection error of the player 3D points to the auxiliary camera.

Therefore, for a sets of world 3D points X_p with $p \in ground$ and a set of 2D points (pixel location) y_q with $q \in player$, the modelview matrix M_{mv} and the projection matrix $M\text{proj}$ are found by minimizing:

$$\min \sum_{p \in \text{ground}} (||X_p - \hat{X}_p||^2) + \lambda \sum_{q \in \text{player}} (||y_q - \hat{y}_q||^2) \tag{3}$$

with

$$\hat{X}_p = M_{\text{mv}}^{-1}\, M_{\text{proj}}^{-1}\, X_p^{\text{NDC}},$$
$$\hat{y}_q = M_{aux} M_{\text{mv}}^{-1}\, M_{\text{proj}}^{-1}\, X_q^{\text{NDC}}$$

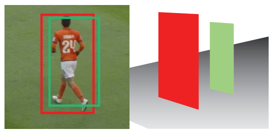
X_p^{NDC} and X_q^{NDC} correspond to the NDC coordinates of ground points p and player points q. The NDC coordinates are found by dividing the the x and y pixel locations with image width and height respectively; the z coordinate is the value of the depth buffer at the specific pixel location. The weight λ was set to 0.01.

Appendix B. Implementation Details

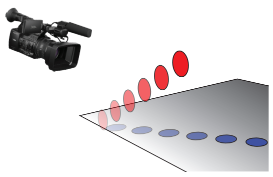
Training Data Our training data comes from the video game Electronic Arts FIFA 2016. The GPU calls between the game engine and the graphics card were obtained using RenderDoc v0.34. The playing teams were randomly selected and the camera was set to broadcast.

Player Analysis Detection and pose estimation was performed using the code of [3] and [5] respectively. Since the boxes can be lifted in 3D, very small or very large detections were removed. For tracking, two tracks were merged if the distance between them is less than 50 pixels and they are inside a frame window of 10 frames.

We observed that a multi-person pose estimator better separates the players than bounding box overlap and it enables a simple algorithm for tracking. The pose estimation skeleton is also used for pixel-wise instance segmentation. In experiments with heavily occluded players, we found the pose estimator was significantly better in separating the occluded players than [6] (94% vs 65% in finding the correct number of players)



Tracking is used for temporal smoothing in 3D, removing jitter, and improving player segmentation. The player mesh is generated from the estimated depth map and the player's 3D bounding box. Small errors in calibration or in the 2D bounding box (the green box shown here is a few pixels off) result in jittering of the 3D box (the assumption is that the bottom of the box lies on the ground) which is corrected using 3D temporal smoothing.



Ball Reconstruction Our method does not reconstruct the 3D position of the ball (in some videos the ball was added manually). Our input is a monocular video and even with perfect 2D tracking of the ball, there is still ambiguity in the 3D ball trajectory that generated the 2D track. For example, we cannot disambiguate whether the ball is airborne moving straight away from the camera (red) or just moving in a straight line on the ground (blue), without incorporating ball physics (an area of future work).

Depth Estimation Network We used the Pytorch implementation of Stacked the Hourglass network [2] with 8 stacks of 1 module. We performed optimization with the Adam solver with 0.0001 learning rate, 0.003 weight decay and betas were set to 0.9 and 0.999. Batch size was set to 6. The network was trained for 300 epochs with cross entropy loss.

We experimented with a number of network architectures: encoder-decoder with skip connections, fully convolutional with upsampling, and others, and we found that the hourglass model had superior performance.

Scene Reconstruction The Soccer Hologram results were obtained using the Microsoft HoloLens Unity SDK and the capture of the video and images were performed using the Mixed Reality Capture from the HoloLens device. The varying-viewpoint results were obtained using Blender, where the reconstructed players were placed in a synthetic stadium with predefined camera paths. No user intervention was required for the players animation.

References

[1] RenderDoc. `https://renderdoc.org`. 1

[2] A. Newell, K. Yang, and J. Deng. Stacked hourglass networks for human pose estimation. In *ECCV*, 2016. 2

[3] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *NIPS*, 2015. 2

[4] S. R. Richter, Z. Hayder, and V. Koltun. Playing for benchmarks. In *ICCV*, 2017. 1

[5] S.-E. Wei, V. Ramakrishna, T. Kanade, and Y. Sheikh. Convolutional pose machines. In *CVPR*, 2016. 2

[6] J. D. X. J. Yi Li, Haozhi Qi and Y. Wei. Fully convolutional instance-aware semantic segmentation. 2017. 2

[7] F. Yu and V. Koltun. Multi-scale context aggregation by dilated convolutions. In *ICLR*, 2016. 1