I will research open-source 3D face authentication models that operate on point cloud input, specifically in PLY format and preferably implemented in PyTorch. I'll gather details on available models, their performance, datasets used, and how to implement them. I'll get back to you with my findings soon.

Open-Source 3D Face Authentication Models (Point Cloud, PyTorch)

3D face authentication (or recognition) models operating on point cloud data have emerged in research, though they are less common than 2D image-based models. A few open-source PyTorch implementations exist that take 3D face point clouds (often in **PLY** format) as input. Below, we identify notable repositories, along with their performance, datasets, implementation details, community support, and how they process point cloud data. We also compare their advantages and limitations.

3DFacePointCloudNet (PointNet++ Based Face Recognition)

Repository & Model: *3DFacePointCloudNet* (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition) (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition) is an open-source PyTorch implementation for 3D face recognition based on the PointNet++ architecture. It was introduced in a 2019/2020 study aiming to learn face embeddings directly from raw point clouds (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition). The model uses a hierarchical PointNet++ network, augmented with a **curvature-aware point sampling** strategy to focus on discriminative facial regions (e.g. nose, eyes) (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition). This approach addresses two key challenges: the lack of large-scale 3D face data and the need for a network that can extract features directly from unstructured point sets (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition).

Performance & Datasets: The authors trained the network on **synthetic 3D face scans** (half a million samples) generated from a 3D morphable model, then fine-tuned on real scans (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition). It achieved state-of-the-art accuracy on public benchmarks **FRGC v2.0** (Face Recognition Grand Challenge) and **Bosphorus**. In rank-1 face identification, it reached **99.46% on FRGCv2** and **99.65% on Bosphorus** (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition), nearly matching or exceeding prior methods. For instance, it obtained a rank-1 of 99.46% on FRGC, very close to the 100% achieved by the best 2019 method (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition). In verification tests (1:1 authentication), it also delivered excellent results – e.g. **99.6% verification rate at 0.1% FAR** on FRGC (GitHub -

alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition), indicating extremely low false acceptance at high true match rates. These metrics show the model's robustness to variations in pose and expression present in those datasets.

Implementation Details: The code is built on an existing PointNet2 PyTorch library (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition). It requires compiling custom CUDA/C++ extensions for point cloud operations (e.g. neighbor grouping, sampling). After cloning the repo, users run `python setup.py build_ext --inplace` to build the extension (producing a `.so` module) (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition). Example training scripts are provided for a two-stage training: (1) **Classification pre-training** on the large synthetic dataset (`train_cls.py`), and (2) **metric learning fine-tuning** with a triplet loss on real face data (`train_triplet.py`) (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition). The model takes raw point clouds as input (each face represented by XYZ coordinates, and optionally normals/curvature as additional features). In data preparation, the authors used the Basel Face Model (BFM2017) to generate faces and the PCL library to compute per-point normals and curvature attributes (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition). This suggests the network can leverage geometric features beyond coordinates. During inference, a face point cloud (e.g. loaded from a PLY file) is passed through the network to produce a 512-D feature vector for that face (), which can be compared via cosine distance for authentication.

Processing Pipeline: Being PointNet++-based, the model processes point clouds in a hierarchical fashion: it samples points to form local neighborhoods and uses set abstraction (with MLPs and pooling) to extract features, progressively reducing the point count while increasing feature dimensionality. The *curvature-aware sampling* modifies PointNet++'s usual farthest-point sampling by prioritizing points with high curvature (facial keypoints) (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition), which helps the network focus on important facial contours. This leads to more discriminative embeddings, as the network learns facial shape signatures that are invariant to expression or other noise. The approach is inherently rotation and translation invariant up to the initial alignment; in practice, faces were likely pre-aligned to a canonical orientation using manual or algorithmic alignment (common in 3D face datasets).

Ease of Use: As a research prototype, 3DFacePointCloudNet requires some expertise to use. Setting it up involves installing PyTorch and building the custom layers. The repository includes instructions and training scripts, but the user must obtain or generate the training data. The synthetic data generation is MATLAB-based (using a provided script and the BFM model) (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition), which could be a hurdle if one just wants to evaluate the model. Pretrained weights are not explicitly provided in the repo (the user is expected to train or fine-tune as needed). However, the fine-tuning stage only needs a small set of real 3D scans (the authors used a subset of FRGC/Bosphorus) to adapt the model (). For evaluation on your own

PLY data, you would need to ensure the point cloud is normalized/aligned similarly to the training data (e.g. centered and scaled). Once set up, inference is straightforward: the model can ingest an Nx3 array of points (read via a library like `plyfile` or `open3d`) and output an embedding.

Community & Development: This repository is academically oriented (MIT-licensed). It has a modest user base (~60 stars on GitHub) (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition). There are only a handful of commits, suggesting limited ongoing development beyond the initial release. Issues and pull requests are minimal or none, implying that community support and continuous maintenance are not strong. Users have successfully built upon it in subsequent research – for example, one paper on a transformer-based 3D face recognizer used the authors' code for comparison (3D Face Recognition: Two Decades of Progress and Prospects). Overall, one should expect to self-troubleshoot environment or data preparation issues. Dependencies include PyTorch (the code was developed around 2019, so PyTorch 1.1–1.4; newer versions may require minor fixes), CUDA for the custom ops, NumPy, etc., and MATLAB + PCL for data generation (only needed if reproducing the training process) (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition). Despite the extra setup, the benefit of this model is its **proven high accuracy** on known benchmarks using raw 3D data, with a well-documented approach.

Advantages: 3DFacePointCloudNet directly learns from raw geometry, preserving 3D information that 2D projections might lose. It achieves excellent accuracy and is robust to facial expression changes (as evidenced by near-perfect rank-1 on the Bosphorus expression subset) (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition). The use of synthetic data is a clever way to overcome data scarcity – it means the model can be trained without requiring a massive real 3D face corpus () (). This lowers the barrier to train a powerful model. Also, by operating on point clouds, it can in principle generalize to any sensor that produces 3D points (structured light scanners, LiDAR, stereo depth, etc.), as long as the input is in the form of a point set of the face.

Limitations: Being a complex deep model, it is relatively heavyweight compared to classical 3D face matchers – inference speed might be a consideration, though PointNet++ is efficient on moderate point counts (the face scans here are on the order of a few thousand points). The requirement of initial alignment or consistent orientation is not fully eliminated; if a face is rotated or upside-down, the network might not correctly recognize it unless trained with such variation. Another limitation is the reliance on an extensive data generation and training pipeline – for a practitioner who just wants to use a pre-trained model on their data, the lack of an easy plug-and-play pretrained model is a drawback. Additionally, while the synthetic-to-real transfer worked impressively, some scenarios (e.g. different sensor noise or extreme occlusions) might still pose a domain gap. Fine-tuning on a small set of real samples is recommended for best performance (). Community support is limited, so updates for newer PyTorch or help with installation issues may not be readily available. Nonetheless, as an open-source reference model, 3DFacePointCloudNet demonstrates how point cloud data can be leveraged for high-accuracy face authentication using deep learning.

SqueezeFace (LiDAR-Based Point Cloud Face Recognition)

Repository & Model: *SqueezeFace* (GitHub - kyoungmingo/Fusion_face: Face recognition using point cloud from LiDAR sensor) is an open-source PyTorch model that integrates 3D point cloud data with 2D image data for face recognition using LiDAR (depth) sensors. It was introduced by Ko et al. (2021) as an "**Integrative Face Recognition**" method for devices equipped with active depth sensors (like the Apple TrueDepth LiDAR on iPhones) (GitHub - kyoungmingo/Fusion_face: Face recognition using point cloud from LiDAR sensor). The architecture is inspired by **SqueezeSeg3**, an efficient CNN originally designed for LiDAR point cloud segmentation in self-driving cars (Integrative Face Recognition Methods with LiDAR Sensors). In SqueezeFace, the 3D point cloud of a face is first projected into a structured format (preserving depth and geometric features), then processed by a convolutional neural network with attention mechanisms (Integrative Face Recognition Methods with LiDAR Sensors). The network also leverages the corresponding RGB image of the face (if available) – hence "fusion_face" as the repo name, indicating **multi-modal fusion** of RGB and depth. Essentially, SqueezeFace converts the sparse point cloud into a **2D depth map with extra feature channels**, which allows applying 2D CNN techniques for face recognition.

Data Input (PLY Usage): The model expects data from an RGB-D capture. In the provided code, the authors use the `plyfile` library to read raw point clouds from PLY files (Fusion_face/dataset.py at master · kyoungmingo/Fusion_face · GitHub) and then save them as `.npy` for training. Each face sample is represented as an image-like tensor, with channels encoding depth and spatial coordinates. For example, they form a 4-channel "depth image" where each pixel corresponds to a point on the face: channel0 = depth (distance from sensor), channel1–3 = x, y, z coordinates (possibly offset and scaled) (Fusion_face/dataset.py at master · kyoungmingo/Fusion_face · GitHub) (Fusion_face/dataset.py at master · kyoungmingo/Fusion_face · GitHub). These are min-max normalized to [0,1] (Fusion_face/dataset.py at master · kyoungmingo/Fusion_face · GitHub). This way, the point cloud (which is essentially an unstructured set of 3D points) is mapped to a fixed-size structured grid. The grid is likely obtained by a cylindrical or spherical projection of the point cloud (as common in LiDAR processing), where the face's point cloud is projected according to the sensor's perspective. The code indicates cropping/resizing to 128×128 in training augmentation (Fusion_face/dataset.py at master · kyoungmingo/Fusion_face · GitHub), which implies the depth images are around that resolution. If an **RGB image** of the face is available (captured simultaneously by the device's camera), it can be fed in as an additional input ("feature_type='full'" in the code uses both RGB and depth features) (Fusion_face/dataset.py at master · kyoungmingo/Fusion_face · GitHub). The network then either concatenates these modalities or uses parallel branches that merge at a later stage (the exact fusion method is not explicitly described in the readme, but attention blocks were mentioned in the paper).

Performance & Experiments: SqueezeFace was evaluated on a real-world setup using an iPhone's TrueDepth sensor. The authors report very high accuracy in their experiments. Incorporating the 3D depth information alongside the 2D image significantly improved performance – the model achieved **99.88% face recognition accuracy** with an F1 score of 93.45%, outperforming a baseline model that used only the RGB images (Integrative Face Recognition Methods with LiDAR Sensors). This suggests that the depth/point cloud data adds

important discriminatory power, likely making the system more robust to lighting or appearance changes. (The baseline was presumably a conventional 2D face recognition method, possibly using a pretrained ArcFace model, as the repo references ArcFace (GitHub - kyoungmingo/Fusion_face: Face recognition using point cloud from LiDAR sensor).) The near-100% accuracy indicates that on the tested dataset (which might have been a limited set of individuals in controlled conditions), the fused model can almost perfectly verify identities. It's worth noting that this result was on the authors' collected data (the paper doesn't cite standard 3D face benchmarks, so they likely created a custom dataset using the iPhone sensor). The focus was on demonstrating reliability in a *practical authentication scenario* (e.g. unlocking a device with face scan). The model's strong performance highlights the benefit of combining **geometry (shape)** and **texture** cues: even if the face is partially occluded or under poor lighting, the 3D shape can still authenticate the user (Integrative Face Recognition Methods with LiDAR Sensors).

Implementation Details: The repository provides code for data loading, training, and testing. It defines a custom dataset class that handles reading the pre-saved `.npy` files (which contain the multi-channel face data) (Fusion_face/dataset.py at master · kyoungmingo/Fusion_face · GitHub). If starting from raw PLY scans, one would use the provided `datagen.py` to convert PLY to the numpy format. The neural network architecture (in `backbone/`) likely builds on a **SqueezeNet/SqueezeSeg** style backbone, which is a light CNN with fire modules (bottleneck layers) and possibly spatial attention. The mention of "attention-based deep conv model" in the paper summary (Integrative Face Recognition Methods with LiDAR Sensors) implies they integrated an attention block to help the network focus on salient face regions. This is plausible since faces occupy only part of the depth image (background pixels would be empty/no points), so attention can ignore irrelevant areas. The training probably uses a classification or metric learning loss similar to ArcFace or Softmax. In fact, references to ArcFace and CosFace repos (GitHub - kyoungmingo/Fusion_face: Face recognition using point cloud from LiDAR sensor) suggest they might use a **margin-based softmax loss** (common in face recognition) to ensure the learned embeddings have large inter-class angular margin. The provided `train.py` and `test.py` scripts allow training from scratch or evaluating the model. The code is relatively self-contained and relies on standard libraries: PyTorch, torchvision (for image transforms), and `plyfile` for data. There is no need to compile custom CUDA code – all operations are done via tensor transforms or high-level libs, which makes it easier to run on different platforms.

Ease of Use: SqueezeFace appears to be straightforward to use, assuming you have access to the required data. For someone with an iPhone or iPad with LiDAR, the authors mention an application (likely an iOS app on the developer's GitHub) that can capture a person's RGB image and corresponding point cloud (GitHub - kyoungmingo/Fusion_face: Face recognition using point cloud from LiDAR sensor). These can then be fed into the model. If using the code as-is, one would collect a set of face scans (each with a PLY and an image), run a preprocessing to generate the `.npy` data, and then run `train.py` to learn a face recognition model on that data. Since the network is lightweight, training on a modest GPU is feasible; the authors reported ~50K parameters (in line with SqueezeNet's small size) and an ability to run in real-time on device. The model is suited for **real-time face authentication** on mobile devices, given its design. The repository includes an example visualization of real data (GitHub - kyoungmingo/Fusion_face: Face recognition using point cloud from LiDAR sensor), showing it

in action. Community-wise, this project is relatively niche – it has only a dozen stars and one fork (GitHub - kyoungmingo/Fusion_face: Face recognition using point cloud from LiDAR sensor). It's likely a one-off research release, without ongoing updates. However, being a simple codebase, it may not require much maintenance.

Processing Pipeline: The key difference in SqueezeFace's processing is that it **projects the point cloud to a 2D representation**. This has pros and cons. The advantage is that standard convolutional layers can be used, which are highly optimized on GPUs. The model effectively treats the face like a range-image (depth map), similar to how the Microsoft Kinect outputs a depth image. This structured approach also makes it easier to combine with a normal 2D face image – one can stack the depth map and the RGB image (plus perhaps an IR image, etc.) in multiple channels or parallel streams. The network likely first processes the RGB image through a CNN and the depth+geometry map through a parallel CNN (possibly sharing the architecture), then concatenates the features for the final identity classification. The use of **attention** helps it learn which modality or region to trust more (e.g. around hair or glasses, the depth data might be noisy, so it can rely on RGB; for regions with low texture but distinct shape, rely on depth). By training on both, the model learns a fused embedding for each face. Notably, this approach implicitly requires the point cloud to be consistently oriented (i.e., the face roughly facing the camera in the depth map). In practice, the device's face tracking ensures that (the user positions their face in front of the sensor). If the face is rotated significantly to the side, some points will be missing; SqueezeFace doesn't explicitly do point cloud alignment or pose normalization – it depends on the sensor's viewpoint. Thus, one limitation is that it may not handle arbitrary rotations as gracefully as a more view-invariant point network (like PointNet++) could.

Advantages: SqueezeFace's multi-modal design makes it **highly accurate in controlled settings**, as evidenced by ~99.9% accuracy reported (Integrative Face Recognition Methods with LiDAR Sensors). The inclusion of depth overcomes challenges that purely 2D methods face (illumination changes, makeup, etc.), since the 3D structure of one's face is hard to spoof or change. It's also efficient – by leveraging a SqueezeNet-derived architecture, it keeps computation low. The model was demonstrated on mobile hardware (iPhone), indicating it's lightweight enough for embedded deployment. In terms of ease of use, it doesn't require a huge training dataset: the authors' experiment used on the order of tens of individuals (the Journal of Sensors paper is only 8 pages, suggesting the scope was a prototype rather than a large-scale study). Even with limited data, the network achieved strong results, which is promising for real-world authentication systems (where you often only enroll one person and possibly train a model to recognize just that person vs. others). The code's simplicity (no custom layers) means it can be extended or modified without low-level programming — for instance, one could easily change the input projection or add another sensor input (like infrared) because it's all high-level PyTorch.

Limitations: The main limitation is that SqueezeFace is tailored to a specific scenario: front-facing depth sensors with an accompanying camera. It **may not generalize** to arbitrary 3D face scans without modification. For example, if given a raw 3D mesh of a face with no associated camera image, the network would lose its RGB input advantage and would need retraining using only the depth channel. Also, the projection approach means it inherits the sensor's perspective – parts of the face not visible to the sensor (e.g. ear regions if slightly turned) are simply missing in

the depth map (sparsed out). The network might misidentify faces if large portions are occluded (though one could imagine training it on various poses to alleviate this). In contrast, a point cloud-specific network could, in theory, be trained to handle partial data by data augmentation (e.g. randomly dropping points). Another limitation is that the method hasn't been benchmarked on standard datasets like FRGC or Bosphorus in the literature, so its performance in more diverse scenarios (different scanners, wider population, more extreme expressions) isn't documented. The **community support** is minimal, meaning if bugs or compatibility issues arise, a user might have to fix them independently. However, since the method builds on well-known components (SqueezeNet, etc.), finding help for general issues is possible (e.g. PyTorch forums for any runtime errors). Finally, while the code can read PLY, the actual training uses a converted numpy format – so a user must perform that conversion step. This is a minor inconvenience but worth noting when setting up.

Other Notable Approaches and Comparisons

Beyond the above two open-source projects, there are a few other research efforts in 3D face authentication that are worth mentioning for context, though their code may not be publicly available. **PointFace** (Jiang et al. 2021) is one such method that directly processes point sets for face recognition using a *Siamese* network architecture (PointFace: Point Set Based Feature Learning for 3D Face Recognition). It employs twin PointNet-based encoders on a pair of face point clouds to learn embeddings via contrastive loss, and reported strong results on datasets like Lock3DFace and CASIA-3D. The approach is similar in spirit to 3DFacePointCloudNet in that it operates on raw points, but it focuses on learning a verification metric directly (good for one-to-one authentication). Another recent work by Wang et al. (2023) introduced a **Siamese PointNet** for 3D face verification under occlusions, showing improved robustness to pose and partial face data. These methods indicate a trend towards using *pairwise learning* (Siamese/triplet networks) to handle the variability in 3D face scans. However, without open implementations, their impact is mostly in inspiration and algorithmic ideas.

In comparison to SqueezeFace's image-fusion approach, some researchers have explored using only depth images for face recognition. A straightforward baseline is to convert a 3D face scan into a **depth map or range image** (for example, rendering the 3D point cloud from a frontal view) and then apply 2D CNN face recognition techniques. This approach leverages mature 2D models but may lose some 3D information. For instance, one could take a pre-trained 2D face recognizer (like ArcFace) and fine-tune it on depth images of faces – but such models would need retraining and are not readily available off-the-shelf. The advantage of keeping data as **point clouds** (as in PointNet++ models) is that the full geometric structure is used and the model can theoretically be viewpoint-invariant (the network can learn to align or ignore rotations). The downside is that point-based networks can be harder to optimize and deploy, and often require more data or augmentation to cover all orientations of a face.

Dependency and Setup Summary: For all these 3D face models, common dependencies include Python (the PyTorch deep learning stack) and a way to handle 3D data (either via libraries like Open3D, PCL, or `plyfile` for reading PLY files). Ensuring your environment can read PLY and possibly visualize point clouds (for debugging) is useful – Open3D is a popular library for that, though not strictly required by the above repos. If using 3DFacePointCloudNet, a

C++ compiler with CUDA is needed for the custom layers; SqueezeFace only needs standard Python packages. Dataset preparation is a significant step: one may need to download academic datasets (FRGC v2, Bosphorus, etc. – typically available to researchers) and convert them to the format expected by the code. FRGC, for example, provides raw scans in a special `.abs` format (a custom binary for meshes/points); converting those to PLY or directly to NumPy arrays would be necessary. The open-source *face3d* repository (GitHub - basilfx/Course-Face3D: Implementation of a prototype 3D Face recognition system) (GitHub - basilfx/Course-Face3D: Implementation of a prototype 3D Face recognition system) (an older academic project in Python) contains some utilities for reading and normalizing such data, though it uses classical methods. In general, using these models for one's own authentication system will involve a non-trivial setup phase, but once configured, they offer a cutting-edge capability of face recognition that exploits 3D information.

Comparison of Approaches

Accuracy: Both open-source models demonstrate that incorporating 3D geometry can yield extremely high authentication accuracy. 3DFacePointCloudNet leads on standardized benchmarks (thanks to its training on massive synthetic data and powerful PointNet++ backbone), while SqueezeFace shows near-perfect accuracy in a focused scenario by fusing depth with images. For applications like secure access or identity verification, these models promise a lower error rate than 2D face recognition in challenging conditions (e.g. low light or attempts to use photo masks) because the 3D shape adds a layer of security (it's hard to falsify the depth of a real face).

Speed and Complexity: SqueezeFace, using 2D convolutions, is likely faster and more lightweight – suitable for mobile or embedded deployment. Its architecture (inspired by SqueezeNet) is optimized for speed. In contrast, 3DFacePointCloudNet, while not unreasonably slow, involves more computation with thousands of points and multiple set abstraction layers; it's geared towards accuracy over efficiency. If run on a GPU, it can still be real-time for one face (PointNet++ can process on the order of 10k points in a few milliseconds), but on CPU or mobile it would be slower.

Data Requirements: A notable difference is data requirements. SqueezeFace doesn't need a large training set if one is fine-tuning it for a particular person or a small set of people (in fact, one could treat it as a feature extractor and just enroll one identity). 3DFacePointCloudNet was built with the assumption of needing huge data – hence the synthetic generation pipeline. If one doesn't have the ability or need to train from scratch, one could use the pre-trained weights from synthetic training (if available) and only do the triplet fine-tune on a small custom dataset. But that still requires at least a handful of 3D scans of each person for fine-tuning. In summary, for a *developer looking to implement 3D face login* for a device, SqueezeFace's approach might be more straightforward (use a smaller model, possibly even without any additional training if the model generalizes well). For a *researcher or company aiming at a universal 3D face recognition engine* that works across datasets and sensors, the PointNet++ approach may be more suitable due to its higher capacity and proven cross-domain generalization when properly trained (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition).

Community and Support: As of now, neither model has a large active community like popular 2D face frameworks do. 3DFacePointCloudNet, being linked to a published paper, might see occasional use in academic circles and could receive unofficial updates (e.g. forks adapting it to PyTorch 1.10+ or adding new data). SqueezeFace is quite niche and tied to the authors' specific use-case. Future developments in 3D face authentication might converge with larger projects (for example, the InsightFace project has begun incorporating 3D face analysis, though mostly for 3D face reconstruction rather than point cloud recognition). If community support is a priority, one might look at general 3D deep learning libraries like **PyTorch3D** or **Torch-Points3D**, which provide modules to build custom point cloud models. While these libraries don't come with a pre-built face authentication model, they make it easier to experiment with architectures (point convolutions, attention, etc.) on 3D data and could be used to reproduce the above models with more modern components.

Advantages vs Limitations: To summarize the trade-offs – using raw point cloud networks (PointNet++ variants) gives **maximal use of 3D geometry** and can be sensor-agnostic, but demands careful training and more computation. Using image-based or projected approaches (like SqueezeFace) yields **computational efficiency** and leverages mature 2D CNN technology, but can be tied to a specific sensor setup and might not utilize the full 3D information if the projection is lossy. Both approaches far exceed traditional geometry-based methods (like ICP matching or hand-crafted 3D features) in accuracy (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition) (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition), thanks to learning discriminative features. However, with great power comes the need for proper data – 3D data is harder to gather than 2D, which is why synthetic augmentation (as done by 3DFacePointCloudNet) or multi-modal input (as done by SqueezeFace) are critical innovations.

In conclusion, open-source point-cloud-based face authentication is an evolving field. **3DFacePointCloudNet** and **SqueezeFace** exemplify two different philosophies: one focuses on *heavy training and 3D feature learning* to achieve universal recognition performance, and the other on *real-time deployment and sensor fusion* for practical authentication. Both demonstrate that with the right processing of 3D point data – whether through direct set processing or clever projection – one can achieve highly accurate face recognition that improves upon 2D-only methods. As 3D sensors become more common (e.g. in smartphones or automotive applications), we can expect more community-driven projects to appear. For now, these open models provide valuable insights and starting points for anyone looking to explore 3D face authentication using PyTorch.

Sources:

- Alfredtorres et al., *3DFacePointCloudNet* – 3D face recognition via PointNet++ (PyTorch code and results) (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition) (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition)

- K. Ko et al., *SqueezeFace* – Integrative face recognition with LiDAR (PyTorch code and paper results) (Integrative Face Recognition Methods with LiDAR Sensors) (Fusion_face/dataset.py at master · kyoungmingo/Fusion_face · GitHub)
- Repository documentation and code for data preparation, network training, and results comparison (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition) (GitHub - alfredtorres/3DFacePointCloudNet: Learning Directly from Synthetic Point Clouds for "In-the-wild" 3D Face Recognition).